Object-oriented Data Mining

Simon Alan Rawles

A thesis submitted to the University of Bristol in accordance with the requirements for the Degree of Doctor of Philosophy in the Faculty of Engineering, Department of Computer Science.

Approximately 76,000 words excluding front matter and bibliography

July, 2007

ALL MISSING PAGES ARE BLANK

IN

ORIGINAL

Abstract

Attempts to overcome limitations in the attribute-value representation for machine learning has led to much interest in learning from structured data, concentrated in the research areas of inductive logic programming (ILP) and multi-relational data mining (MDRM). The expressiveness and encapsulation of the object-oriented data model has led to its widespread adoption in software and database design. The considerable congruence between this model and individual-centred models in inductive logic programming presents new opportunities for mining object data specific to its domain.

This thesis investigates the use of object-orientation in knowledge representation for multi-relational data mining. We propose a language for expressing object model metaknowledge and use it to extend the reasoning mechanisms of an object-oriented logic. A refinement operator is then defined and used for feature search in a object-oriented propositionalisation-based ILP classifier. An algorithm is proposed for reducing the large number of redundant features typical in propositionalisation. A data mining system based on the refinement operator is implemented and demonstrated on a real-world computational linguistics task and compared with a conventional ILP system.

Keywords: Object orientation; data mining; inductive logic programming; propositionalisation; refinement operators; feature reduction.

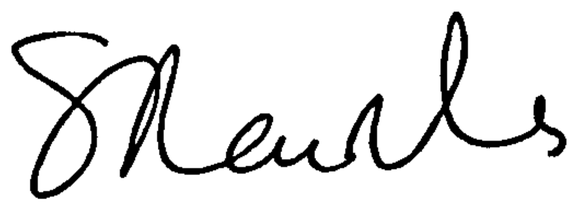
Acknowledgements

There are many people who have helped me along the long road to writing this thesis, without whom it would never have been produced. I would like to thank my supervisor, Peter Flach, who not only got me interested in machine learning in the first place but who patiently taught me how to be a better academic, faced my demons with me and was still there beside me when I eventually drew that line in the sand. I have been lucky enough to have done this research as part of the Machine Learning and Biological Computation group at Bristol University over the last five or so years, and I thank them for all the ideas, advice and support. Thanks particularly to: Elias Gyftodimos, Tim Langford and Shaomin Wu for showing me the way of ILP when I was new to it; Hsiou-Wen Hseuh for being a great gym partner and cook; Stuart Reynolds and Tim Kovacs for not only machine learning help but film nights and crash space; Simon Price for his enthusiasm and regular M5 shuttle service; Rob Egginton for being a great guy to share a desk with; Tarek Abudawood for his ceaseless enthusiasm and lots of ideas and insights, even at 3am; Susanne Hoche and Ksenia Shalonova for cheerfully wading through drafts and happily answering a neverending stream of questions on ILP and Computational Linguistics; Annalisa Appice and Michelangelo Ceci for making Boolean matrices something to get excited about; and finally to James Marshall and Rafal Bogacz who never tired of me visiting their offices for ideas and advice.

There are many other people who have helped me in less academic ways. Of course, the biggest thanks must go to my mum, dad and Hannah, for always being there and providing support in more ways than I could have ever expected of them. I've also been lucky to have many good friends who have seen me through the many ups and downs of this project, provided important distractions during the writing-up stage and known when the right and wrong times were to ask 'have you done your thesis yet?'. Among them: Jim Farrand, Sharon Shadrokh, Julian Brown, Jamie Shilton, Nugget Smith, Jon Pike, and the other members of the Bristol Geek Massive; Deepa Shah, Tim Hutton, Rick Dickerson and the other members of the Tongs for all the klau; Hugh Barnes, Graham Searle and the rest of the Warwick bunch; Steve Bright, for showing me that surfing provides an excellent distraction; Henry Nelson, Jeff Belcher, Chris Brown and Ralf Doswell for making Coventry a fun place to run away to; and to Nina Jenkin for her positive encouragement and for many happy adventures.

Declaration

I declare that the work in this dissertation was carried out in accordance with the Regulations of the University of Bristol. The work is original, except where indicated by special reference in the text, and no part of the dissertation has been submitted for any other academic award. Any views expressed in the dissertation are those of the author.



Simon Rawles

July 5, 2007

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

This chapter presents the motivation for the work described by the rest of the thesis. It comprises three parts. In section 1.1, the preliminary concepts and terminology relating to multi-relational data mining and object-orientation are presented. Section 1.2 situates these concepts in the field of inductive logic programming. Section 1.3 considers the application of object-orientation to machine learning. Section 1.4 presents the aims of the work describe in this thesis and describes the overall structure of the thesis, and section 1.5 concludes.

1.1 Knowledge discovery in databases and data mining

The highly multidisciplinary field of *knowledge discovery in databases* (KDD) is concerned with the discovery of patterns in data. It may be defined as "the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data" [50]. Interest in KDD has grown rapidly over the last ten years as the acquistion and analysis of massive data sets has become increasingly important to the operation of a wide variety of organisations.

Process models defined for KDD, such as CRISP-DM [129], broadly agree on the stages involved in KDD. Firstly, a data understanding stage (possibly including a data collection stage) takes place in order to gain familiarity with the data and identify issues. Next, the data preparation stage transforms the raw data into a form suitable for computational analysis. This analysis phase then begins, and the patterns are identified through a data modelling process. The discovered patterns, the results, are finally evaluated and interpreted. KDD has a wide variety of industrial applications, from fraud detection to bioinformatics, from decision support to marketing and from retail to counter-terrorism. *Data mining* (DM) may be viewed as a (semi-automated) stage within this knowledge discovery process, namely the central pattern-finding stage. One practical definition of data mining is given by Fayyad [42] as follows:

> Data mining is a step in the KDD process consisting of applying computational techniques that, under acceptable computational efficiency limitations, produce a particular enumeration of patterns (or models) over the data.

Many of these computational techniques can be said to be computer programs capable of *machine learning* (ML), that is, such a program learns from experience E with respect to some class of tasks T and performance

measure P, if its performance at tasks in T, as measured by P, improves with experience E [97]. Informally, the term 'machine learning' describes a class of programs which perform better with experience.

In *single-relation data mining*, the information that the data miner, or learner, is given as a *data set* which is made up of a set of *instances* or alternatively *examples*. The data are taken from a context, referred to as the *domain*. *Domain experts* may be consulted to gain *domain knowledge* which may be incorporated into the data mining process. Each instance consists of a series of *values* assigned to the same set of *attributes* (or alternatively *features*) that describe some aspect of the instance. Attributes may take on different types; they are typically either discrete, taking their value from a set of possible values, or continuous, taking on a real number, perhaps from some defined range. These examples are said to be in *attribute-value* or *propositional* form. As will be discussed later, this is not always the most natural way of representing data, but it is by far the most common.

The process of data mining and finding patterns often involves the formation of something to be learned, which we call the *concept* (or alternatively the *model* or *theory*), often a generalised description of the instances, describing a set of patterns within those instances. While a concept typically classifies an example according to whether it belongs to a target class or not, more general formulations exist for problems involving more than one possible target class. The specific form of the concept depends heavily on the choice of learning algorithm. The process of generalising from a set of instances to a concept is called *induction*.

We may characterise learning as the discovery of patterns in the data, which leads us to consider the form of these patterns and how they are expressed. The patterns can take many forms and are highly dependent on the choice of the data mining task and the learner chosen for it. Witten and Frank [139] categorise the learning tasks typically used in data mining into four basic styles. *Classification learning* involves learning to classify unseen examples based on already classified examples. In classification learning one attribute in the dataset is identified as the *class* of the instance. Many learners have been proposed, including those that produce decision trees (ID3, C4.5 and the widespread C5.0), decision lists, Bayesian models (Naive Bayes, Bayesian Networks), classification rule learners (Ripper, CN2), instance-based learners (nearest-neighbour methods), and support vector machines. *Numeric prediction* involves predicting some numeric quantity rather than a discrete classification. Models include regression trees, artificial neural networks, and traditional methods such as linear regression and least mean squares approaches. These classification learning and numeric prediction tasks may be grouped together into the category of *prediction tasks*.

Association rule mining is similar to classification rule mining, but seeks to find more general patterns than only predictive ones, in which the head of the rule (the then-part) may contain any number of values for attributes. Association rules algorithms find elements which co-occur frequently within a dataset, according to their *support* (the number of instances they predict correctly) and their *confidence* (the proportion of instances they predict correctly). *Clustering* involves grouping examples together according to some definition of similarity. These groups may overlap or may be mutually disjoint.

Finally, an important aspect of data mining is evaluating the models produced in a mining task. The quantative aspects of predictive data mining models may be evaluated through a wide variety of established and well-defined *data mining metrics*, for example, predictive accuracy and weighted relative accuracy, recall and precision, and the F-measure. Furthermore, predictive performance may be expressed using ROC analysis techniques. Other aspects of the data mining model, such as its understandability or semantic validity, are less conducive to measurement.

1.1.1 Multi-relational data mining

Until this point we have considered data mining in a single-relational world. The data was assumed to occupy a single table, each row an example containing a fixed number of values for attributes, one for each column. However, data is not often expressible in this form in a natural way. For example, suppose that the data mining task operates in a domain concerning molecules, made up of any number of atoms. Since the number of atoms per molecule is variable, we cannot easily define a number of attributes in which to describe every molecule. There are a number of ways in which we can incorporate data into the single-relation representation:

- Each row represents a molecule, and the attributes for each atom are described in sequence. In other words, after writing the values for attributes concerning the molecule, we write the values for the first atom, followed by the second atom, and so on, up to a maximum number of atoms. This is far from ideal. An arbitrary order has been imposed on the atoms, and furthermore, induced models will refer to attributes at a specified point in the sequence.

- Each row represents an atom, and the attributes for each molecule are duplicated across atoms, so each atom is also described in terms of the attributes of the molecule it belongs to. Information is there but there is redundancy, and data mining will give us information about atoms, rather than molecules, which are actually of interest.

- The information about the atoms as a group may be summarised in one or more aggregating statistics. Though this overcomes the problems of the previous two approaches, it inevitably introduces some information loss, and the choice of the aggregration methods necessarily limit the learning task and the forms of patterns that can be discovered.

Because of this restriction, algorithms capable of mining more flexible data representations have been proposed. *Multi-relational data mining*, also called *relational data mining*, is data mining applied to the data in a more flexible form. This *relational* representation can be thought of as a set of tables, each one often relating to some class of objects of interest in the domain being modelled, which are interlinked using a system of identifiers, or *foreign keys*. Relational representations have been used as the basis for mainstream database technology for many years now and have been proven flexible enough for a huge variety of applications.

Multi-relational data mining is related to the mining of data which has structure of its own. As is shown in the example above, not all real-world data can be expressed naturally as a tuple of constants — data structures such as sets and multisets, lists, trees, graphs and spatial representations more accurately model the semantics of the data. We call this kind of data *structured data*, and the multi-relational representation it demands is better handled by MRDM techniques. Section 1.2 discusses techniques for multi-relational data mining.

1.1.2 Object orientation

The term *object-oriented programming* was coined by Alan Kay in connection with the pioneering 1970s object-oriented language Smalltalk [60]. Though its roots were in simulation software, organisations adopted object-orientation for a variety of reasons: it reduces the risks in the development of complex systems, the expressiveness of the language allows the development of models more congruent with human understanding of domains, and it facilitates reuse of established code and models.

An *object* is a collection of typed data, together with the methods which are defined to operate on it. An object-oriented system contains at any stage a set of these objects, which communicate with each other by passing messages[1]. In this way, the data is *encapsulated* — it is not directly accessible to anything other than the methods with which they are accessed. Objects are instantiations of at least one *class*, which can be thought of as a definition of an abstract data type. A class is a template defining the data structures and methods for objects. Objects can also be said to belong to a class.

Broadly, classes can be related in two ways. The first is *composition* or *aggregation*. This (possibly recursive) relationship links many parts to its whole. As was mentioned above, data in objects are associated with types, and it is through this mechanism that composition is implemented — if a class A has data of a given class B, we say that A contains B. It could also contain more than one object of class B, and relationships are characterisable by whether they are 1-to-1, 1-to-n, or m-to-n, the *multiplicity* of the relationship.

The second way in which classes can be related is *inheritance*. Inheritance refers to a relationship among classes where classes share structure or behaviour — a *subclass* inherits the structure and behaviour of its *superclass*, this subclass/superclass relationship introducing a *is-a-type-of* relationship over the classes. In languages with a *metaclass*, all classes inherit the structure and behaviour of this metaclass. The power of inheritance comes from the possibility to inherit a class and add to its behaviour and structure.

These two relationships form the classes into two hierarchies, the *composition hierarchy* and the *inheritance hierarchy*[2]. By combining aggregation and inheritance hierarchies we get the expressive power of object orientation. McCabe [93] calls this the 'dual structure of knowledge'.

1.2 Multi-relational data mining using logic

Having introduced the fundamental ideas of multi-relational data mining and object-orientation, this section presents an overview of multi-relational data mining techniques and the descriptions of the data — the *domain descriptions* — which they use. In section 1.1.1 the concept of multi-relational data, as opposed to attribute-value data, was introduced. This section discusses approaches to mining this data.

Inductive logic programming (ILP) is an approach to multi-relational data mining in which logic programs, consisting of first-order rules, are learned from a set of examples and background knowledge. These examples and background knowledge are expressed in the form of a logic program, in which the concept is based on first order logic. This is one of the distinguishing features of the approach.

Inductive logic programming has the paradigm of logic programming at its foundation. Logic programming is the use of mathematical logic for computer programming. It uses declarative sentences, or clauses, of the form $h \leftarrow b_1 \wedge ... \wedge b_n$ representing implications that where all the body subgoals b_i are satisfied, the head goal h is also satisfied. The logic programming language then solves a query in h by attempting to prove each b_i. The programming language Prolog performs this process on Horn clauses, where h and b_i are all atomic predicate logic formulae and uses a method known as SLD-resolution to solve a query in h. The goals h and b_i may take variables in Progol, lifting it from a propositional setting to that which corresponds closely with first-order predicate logic. In performing inductive logic programming, we take a database of facts describing a set of examples and aim to derive a logic program — a set of clauses — which prove the examples.

---

[1] In practice, this takes place by one object calling a method from another object.

[2] Some languages allow a class to inherit from more than one other. In this case there is an *inheritance lattice* instead.

The difference between multi-relational data mining and inductive logic programming is that, whereas multi-relational data mining typically assume that the data is expressed in a series of interlinked relational database tables, ILP maintains a logic programming view. As Džeroski and Lavrač [38] describe, database and logic programming terms have a close equivalence. This basis in first-order logic provides the approach with an inherent ability to manipulate and reason with structured data. This is particularly useful for the introduction of background knowledge; ILP systems provide a convenient and expressive way to incorporate complex background knowledge into the data mining task through the first-order representation.

More formally, the typical empirical ILP setting is concerned with learning a first-order rule for a binary target relation p, and is given in [77]:

Given:

- A set of training examples $\mathcal{E}$, consisting of true $\mathcal{E}^+$ and false $\mathcal{E}^-$ ground facts of an unknown predicate p,
- a description language $\mathcal{L}$, specifying syntactic restrictions on the definition of predicate p, and
- background knowledge $\mathcal{B}$, defining predicates q_i (other than p) which may be used in the definition of p and which provide additional information about the arguments of the examples of predicate p.

Find:

- A definition $\mathcal{H}$ for p, expressed in $\mathcal{L}$, such that $\mathcal{H}$ is complete and consistent with respect to the examples $\mathcal{E}$ and background knowledge $\mathcal{B}$.

We say a hypothesis *covers* a set of examples if it is true for all examples in that set. A hypothesis $\mathcal{H}$ is *complete* with respect to the background knowledge $\mathcal{B}$ and examples $\mathcal{E}$ if all the positive examples $\mathcal{E}^+$ are covered. Likewise, $\mathcal{H}$ is *consistent* with respect to $\mathcal{B}$ and $\mathcal{E}$ if none of the negative examples $\mathcal{E}^-$ are covered. For noisy domains these criteria may be relaxed.

Constraining the set of hypotheses considered is important to reduce computational complexity and to avoid overfitting — there is an inevitable complexity/expressiveness tradeoff. This forms the *bias* of the learning system, constraining the hypothesis space (the *language bias*) as in the syntactic restrictions described above, or the way in which it is searched (the *search bias*). Similar to traditional concept learning, inductive logic programming can be viewed as a search of the hypothesis space, usually first-order clauses, restricted by an appropriate language bias and search bias, for a complete, consistent hypothesis.

This hypothesis space, made up of logic program clauses, must be structured in some way in order for it to be searchable. ILP systems order the clauses according to a generality relation known as semantic generality, in which a clause c is at least as general as another clause c' if c logically implies c' with respect to the background knowledge $\mathcal{B}$ (written $\{\mathcal{B} \cup c \models c'$). This ordering then introduces a *lattice* over the clauses. In practice, testing for logical entailment is undecidable, and so it is replaced with various techniques for *syntactic generality*, which depend on the provability of c' from c (written $c' \vdash c$) in some adopted proof method. One such technique is θ-*subsumption*, in which if there is a substitution θ assigning terms to variables in a clause such that $c\theta \subseteq c'$, then c θ-subsumes c' [113]. This means $c \models c'$ but the reverse is not necessarily true. The

least general generalisation (*lgg*) of two clauses may then be defined as the least upper bound of the clauses in this lattice. A *specialisation operator* (or *refinement operator*) operates under θ-subsumption to return a set of (usually minimal) specialisations of an input clause, usually by applying a θ-substitution to the clause, or adding a literal to the clause's body. Using this operator, an ILP system can navigate the generality lattice in its search for an appropriate clause.

The search over this lattice is bounded below by the most specific clause covering each example, constructed either as the relative least general generalisation, the most specific inverse resolvent, or by inverse entailment (as in the PROGOL [101] system). The *relative least general generalisation* (or *rlgg*) of two clauses is the least general clause more general than both of them *with respect to background knowledge* $\mathcal{B}$. The ILP system GOLEM [105, 106] uses rlggs in its construction of the 'bottom' bounding clause. *Inverse resolution* inverts the resolution rule found in deductive mechanisms. The ILP system CIGOL [103] uses this technique using generalisation operators which produce inverse (θ-)substitutions.

We can draw a distinction between methods which progressively specialise from more general clauses and those that progressively generalise from more specific clauses. The former are *top-down* techniques, for example FOIL [117] and PROGOL [101], both covering approaches searching the θ-subsumption lattice. FOIL progressively adds literals to the clause by 'variablisation' of predicates and use of built-in predicates, using information gain as the search heuristic, while PROGOL uses an A* heuristic, bounding the search as described above, making it in a sense a hybrid system. The latter are *bottom-up* techniques. These include GOLEM [106], in which each covering step involves the selection of two examples, to which the process of inverse resolution is applied, yielding the most specific inverse subtitution. The two clauses are combined using rlgg and the best clause chosen before the next covering iteration. CIGOL [103] is another bottom-up leaner based on inverse resolution. Attempts have been made to combine top-down and bottom-up induction, for example the CHILLIN system [143].

A final alternative approach is *propositionalisation* [74, 72], the technique adopted for learning in this thesis. Propositionalisation involves a transformation algorithm which generates a set of first-order features, transforming them to attribute/value form by solving them for each example, instead of searching the first-order hypothesis space directly. Learning is then carried out by a conventional attribute-value learner. Examples of propositionalisation learners include SINUS [74] and its ancestor LINUS [78], RSD [84] and RELAGGS [75].

This short overview has considered ILP in the setting of predictive induction — the learning of classification rules. However, some variants exist. *Descriptive induction* seeks to describe and summarise observations rather than predicate a target relation, for example in frequent pattern discovery and clustering. The TERTIUS [47] learner and the WARMR [36] frequent pattern discoverer are both examples of descriptive ILP.

We have already termed the context in which the data mining takes place as the *domain*. We consider the notion of a domain and its modelling for ILP. The idea of *domain analysis* was introduced by Arango [6] as "an attempt to identify the objects, operations and relationships domain experts perceived to be important about the domain". In data mining, and particularly with structured data involving complex data relationships, it is beneficial to have an algorithm which can take advantage of the data relationships and the domain model generally. We communicate this domain model with a *domain description*, written in a suitable *domain description language*. Many ILP systems already incorporate the use of simple domain descriptions, in the form of predicate mode declarations, as in ALEPH [132] or as predicate definition files, as in the first-order Bayesian classifier 1BC [46]. We consider the form of these descriptions in more detail later.

In recent years, data models for multi-relational data mining have used first-order representations with a

strong concept of the individual. They typically operated in a data domain characterised by the *individual-centred representation* [45], an aggregated description of an example. Individuals are represented by an arbitrary hierarchy of typed complex terms such as tuples or lists, and associated with these types are operations available to the learner for the type's decomposition (*structural predicates*) and attributes (*property predicates*). As such, they relate directly to database models, for example the entity-relationship (ER) model [15], which defines relationships among objects in the domain (represented by foreign keys among the tuples of the database) and the attributes associated with them. These objects are intrinsically typed by the model. By identifying one type as the individual, we can consider these relationships to adopt a tree-like[3] structure with the type of individual at the root. The tree involves either one-to-one or one-to-many relationships from the root to the leaves. The types then guide the induction process, constructing hypotheses whose terms interact appropriately to the model definition. Individual-centred representations are immediately useful since they occur naturally in classification problems and are easily specified by a domain expert and allow learning to be more domain-centric. In practical situations the approach allows us to establish a level of complexity which is neither too expressive (leading to overfitting and performance problems) nor too restricted.

1.3 Applying object-orientation to machine learning

We are now in a position to define what is meant by the term *object-oriented data mining* (*OODM*). A possible definition is 'data mining on data expressed in an object-oriented (data) domain model'.

The object model in this thesis may be seen as an extension of the individual-centred representation approach. The individual-centred representation can be situated in the object model by considering the types as classes, the relationships as relationships between objects and the properties as attributes of objects. Remaining elements of the object model present further possibilities for domain description which may be exploited by an induction system. The object data model allows the specification of complex data objects with an arbitrary level of complexity, according to flexible but strong typing rules. The simple types of the individual-centred representation and other means of describing domains are extended in the object model by the notion of a class. The inheritance hierarchy enables the definition of properties of the data model not possible in conventional domain description. For example, it makes sense to describe the engine capacity for a car, but not for a bicycle, even though they are both types of vehicle. The object model allows us to arrange these classes hierarchically (cars and bicycles are both types of vehicle), specifying the engine size property in the car class only. In a conventional representation, these would either occupy the same type, producing possibly large numbers of attributes which are not meaningful for many objects, or as three separate types, in which case the fact that some are types of others is lost on the learning process. Object domain descriptions also allow us to define, for example, that no object is of class vehicle only (the abstractness property) and that no vehicle can be both a car and a bicycle (the disjointness property). The learner's search can be guided more effectively in the light of this additional knowledge. The object-oriented framework is a natural way to express complex objects such as sets, lists or trees. Properties and substructures in these classes of objects can be further exploited by the learner. Moreover, object models provide a natural representation for a number of aspects of the complex relationships occuring between classes and objects in a given domain. For example, where two objects are linked, the link can be specified in terms of its multiplicity and direction. Again, the learner can take advantage of knowledge

[3]The structure is *tree-like* because types are permitted to relate to themselves by one-to-one relationships.

characterising these relationships to better guide its search.

The very large amount of interest in the software engineering and database domains have led to the development of established and proven methodologies for the decomposition of real-world domains into appropriate object models. Similarly, efforts to perform deduction on object-oriented databases have provided robust formalisms born out of object-oriented and logic systems. In the case of inductive logic programming, these formalisms form the basis for the first-order hypothesis language. These are much more fully discussed in chapter 2.

Finally, we identify a number of expected benefits of the object-oriented approach to multi-relational data mining compared to traditional approaches, as follows:

- *The object model offers features supporting the separation of domain and algorithm.* The incorporation of domain knowledge is an important task in many multi-relational data mining tasks. Traditionally this is done incrementally, by adapting the algorithm and the data, adding background knowledge, and so on. An object-based domain description would encapsulate this knowledge in a simple module, for incorporation into a modular framework.

- *The object model introduces practical benefits for data miners.* The capture of the domain in a portable, modular object form will be of benefit to those mining in domain-specific tasks. These include aspects such as ease of reuse and dissemination and the standardisation of interfaces across domains and programs.

- *The object model has a high degree of expressive power and understandably corresponds to the real world.* Object-oriented domains capture complex but relatively human-understandable aspects of real-world scenarios. This facilitates not only the domain analysis process, but makes the resulting models more comprehensible, since they are more closely described in terms of the domain.

- *Object-orientation is a natural extension of existing methods.* The tools of a multi-relational data miner — algorithms and their underlying logics, logic programming, database systems, and the structured data itself — has in many cases a natural extension to the object model. We have seen the relationship with the existing individual-centred representation and further consider the extensions in chapter 2. Furthermore, established and proven methodologies exist for analysis of object-oriented domains from diverse real-world situations.

1.4 Aims and structure of the thesis

It is now possible to make a statement of the research problem motivating this work and to arrive at the aims of the thesis. The general issue being addressed concerns the practical benefits in applying the principles of object-orientation to knowledge representation for multi-relational data mining. Accordingly, evaluation is undertaken using a data mining system capable of natively handling object-oriented data domains. In particular, this system should be a fully-functional, open-source toolkit allowing the user both to analyse and capture the domain and mine the data. With this in mind, in this thesis we aim to show the following.

> By adopting the object model as a means of representing and describing data for arbitrary structured domains, and in particular those adopting individual-centred representations, it is possible

to perform classification rule learning by propositionalisation more efficiently, in a less logically-redundant hypothesis space, producing theories with better predictive performance than state-of-the-art inductive logic programming techniques under comparable parameter settings. Furthermore, we argue that the object model is a natural extension of existing methods, and leads to a more practical separation of data from algorithm.

To clarify the terms used:

the object model refers to a standardised means of describing data which adopts core aspects of defining and describing data from the analysis and design of object-oriented databases and programming languages. The resulting description, the domain model, is defined in a suitable language of constraints and other domain knowledge.

arbitrary structured domains refers to the application of *any* multi-relational database which describes a set of examples which can be expressed in a series of database tables representing typed individuals from the domain, linked via indentifiers for these individuals. Such individuals typically are naturally expressed as trees or graphs.

individual-centred representations means data domains with a strong notion of typed individual, in which component individuals are represented by links between their identifiers and properties are defined on those individuals. We extend the notion of the representation presented in section 1.3.

possible means that by using an implementation of the algorithm in a standard computer language, we can run the program in a reasonable amount of time on standard computer equipment such as desktop PCs.

classification rule learning tasks refers to the induction of classification rules, which attribute one of a set of classes to unseen examples based on a set of provided examples labelled with a class.

propositionalisation refers to the transformation of hypotheses into a single table form for the purposes of learning by a conventional attribute-value learner.

more efficiently means the search undertaken by the inductive learner involves the consideration of fewer candidate hypotheses than under existing approaches, and in particular ignoring hypotheses which are rejected under the quality criteria of the existing approach.

less logically redundant means that the space of hypotheses searched by the object learner contains fewer hypotheses which are syntactically equivalent — that is, those that can be found to be equivalent under variable renaming, literal reordering, *etc.*— or which are otherwise equivalent under some metaknowledge provided in the domain model.

better predictive performance means that the resulting theories possess a higher level of predictive power. Historically, predictive accuracy, the proportion of correctly-classified unseen examples, is the de-facto measure. Measures from ROC analysis [52] often give a better idea of predictive performance and are adopted for this work.

state-of-the-art inductive logic programming technique means an approach to inductive logic programming and its implementation, which is adopted widely by the machine learning community and represents a recent and accepted scientific technique, suitable for use as a benchmark.

comparable parameter settings means that as far as possible, the parameter settings of the compared learners are equivalent to each other under some shared meaning.

natural extension refers to the adoption of object-orientation as an extension to the inductive logic programming technique, both in terms of its data representation and its domain description (bias).

separation of data from algorithm refers to the implementational independence of the database and its object metaknowledge and the algorithm itself. As a result, modular domain libraries specific to the domains under consideration are to be defined and implemented.

Furthermore, we establish two subsidiary aims.

Feature reduction: Large feature sets, a common result of propositionalisation, often negatively impact the predictive performance and running time of learners. Post-processing this feature set using a method which constructs a set partition of the example set and applies an efficient subset-based logical redundancy criterion, yields a subset of features which preserve the learnability of a complete, consistent hypothesis and alleviates these shortcomings.

Grammatical labelling application: The object-oriented data mining system presented can be used to model the complex structures in English-language documents, and give superior performance according to the core aims when compared with an existing inductive logic programming system.

1.4.1 Structure of the thesis

The remainder of the thesis is structured as follows:

Chapter 2 describes the object model in depth, discussing how it can be represented with current approaches to deductive databases and forms of inductive bias, discusses special-purpose logics and adopts one — F-Logic — as the basis for deduction in this thesis, presenting its syntax and semantics.

Chapter 3 expands upon F-Logic to arrive at a logical language for deduction and a bias for induction suitable for learning. It defines desirable properties of an object clause, the additional knowledge introduced by the class structure in the form of constraints and how *metaknowledge* augments the object model.

Chapter 4 discusses the process of induction — the discovery of general properties from specific examples. It considers refinement operators — one clause construction technique in ILP — as a means to search a space of clauses, and proposes a new refinement operator tailored to the requirements of the object model, and particularly, adopting class as the means of constraint on the values taken by terms in the logical language.

Chapter 5 considers how the features constructed during the search process are used in the construction of rules and theories. It presents propositionalisation, an approach to ILP which transforms a first order individual into a sequence of Boolean values so that learning may be performed by an attribute/value learner. In practice, propositionalisation techniques often produce a large number of such features, and a general and efficient algorithm for removing features which are logically redundant in the presence of others in multi-class problems is presented and analysed.

Chapter 6 details the implementation of CoSinus, a full-featured ILP system making use of the refinement operator and feature removal techniques. The chapter expands on the induction techniques from chapter 4, considering a three-tired approach to theory construction at the feature, rule and theory levels. Practical issues of feature, rule and theory construction, including search bounds and treatment of class constraints on terms, are discussed.

Chapter 7 presents a real-world application of inductive logic programming in the object model. A learning task is identified in the field of computational linguistics based on a highly structured, preprocessed corpus describing logical and surface structure of English sentences. Comparisons with a benchmark ILP algorithm — Progol — are made with the aid of a representational and inductive bias mapping between the object model and logic programs and type and mode declarations in a conventional inductive logic programming setting. This comparison includes data mining experiments which demonstrate properties of the search undertaken by the learner and against which the claims above may be verified.

Chapter 8 brings the main points of the thesis together and identifies future work.

1.5 Conclusion

This chapter has situated the research presented in the remainder of this thesis in terms of its general fields. The processes of knowledge discovery and data mining were introduced and the main approaches to data mining were considered. We then considered (multi-)relational data mining, which can be thought of as a class of data mining techniques which aim to find patterns in a set of interlinked database tables. The notion of object-orientation and some elements of the object-oriented data model were introduced. Section 1.2 described these ideas in more depth, introducing the field of inductive logic programming and specific approaches to it. Important concepts such as bias, generality and search were considered. We also characterised the context of an ILP learning task as the domain, and in particular examined a convenient representation — the individual-centred representation — which is of particular relevance to the object model. Section 1.3 considered some expected benefits of applying object-orientation to the data mining process. In section 1.4, the motivating research problem was presented and the remaining chapters of this thesis were outlined.

Chapter 2

Representing object databases in logic

Multi-relational data mining encompasses techniques and methods which extract knowledge from relational databases. Inductive Logic Programming is a research area which involves the study of multi-relational data mining using data expressed as a logic program. Such programs are sometimes termed *deductive databases*. Deductive databases have considerable correspondence with existing relational databases but offer the reasoning mechanisms associated with logic programming.

Object-oriented data mining techniques aim to analyse object-oriented databases in order to discover interesting regularities or patterns among the data. We study the intersection between object-oriented data mining and inductive logic programming as data mining in deductive databases. In such a setting, these regularities will be typically expressed in terms of a logical object language. Particularly of interest to this thesis is the learning of classification rules from such databases. Accordingly, in this chapter we study the use of object-orientation for knowledge representation, with regard to the role of objects in logic, their representation in the database and mechanisms for reasoning using rules. Particularly, we view an object-oriented deductive database (OODD) as an extension of the traditional logic program deductive database, with a view to adapting existing ILP approaches to data mining in OODDs.

This chapter begins by defining what is meant by an object-oriented data model and examines its components. It then studies existing special-purpose logical languages which support aspects of object-orientation, as well as the extent to which Prolog, the prevalent logical representation for ILP, permits object-like representation and reasoning. Next, we focus on F-Logic, a prominent object logic, and study its syntax and semantics, with a view to adapting it to a language for object-oriented data mining in the following chapter.

2.1 The general object model

It is common to define object-orientation by defining a number of concepts which the system must possess. Indeed, there has been a lot of debate in the literature as to what exactly an object-oriented database comprises. Opinions vary greatly, but in general, we can identify core features of object-based systems. The object data model possesses the following characteristics:

Object. The object is the primary data unit, and is intended to refer to some 'real-world' object in the domain being modelled. As identified by Kifer [66], this aspect of the data representation paradigm is the key distinction between object-oriented and relational languages. Relational models spread information regarding

an object across many relation, grouping data where it shares a relation, whereas object-oriented systems group data by object. A symbol in the language, the object's identifier, then acts as a handle on this grouped data.

Class. Each object belongs to a class, the unit of definition of each object's data and behaviour and the basis of the typing mechanism. Arguments and variables are said to be *classed* where they are restricted to belong to a given class. It is the basis of modularity and structure, and typically refers to some recognisable aspect of the problem domain. A class may be seen as the encapsulated combination of an abstract data type and its implementation, specifying data and the operations which the class knows to carry out [49]. Classes thus provide a standard interface independent of the underlying data structure, providing abstraction and encapsulation of data. Note that it is necessary to establish a clear terminological distinction at this stage, since the notion of class may also refer to the label of an individual in a dataset to be learned by a set of classification rules. Where ambiguity arises, 'object class' will be used to refer to the object model sense, and 'class label' for the classification sense.

Properties and methods. By analogy to object-oriented programming languages such as Java, the compositional, or 'part-of', aspect of object oriented data is represented by extensional properties and intensional methods. Multi-relational data mining necessarily operates on data which has structure, expressed in the object model in terms of the properties and method definitions. Properties embody named associations of an object to a composite object. For a given object, its composite object is classed by a signature, but in the general case, the presence of a property does not necessary mean that composite objects are defined, allowing the representation of partial data. Method calls generalise properties, defining a computable result for classed input arguments.

Inheritance. There is a hierarchy of classes such that the *subclass* is defined to be a specialisation or extension of the definition of its *superclass*. Accordingly, such a relationship is sometimes termed a *generalisation* relationship. This hierarchy, often termed the 'is-a' structure, represents a second key structure in object-oriented data, the inheritance structure, and the object model then consists of the two hierarchies of composition and inheritance. Booch [11] observes that "aggregration permits the physical grouping of logically related structures, and inheritance allows these common groups to be easily reused among different abstractions". Where two classes partake in a superclass/subclass relationship under inheritance, all method and property definitions are propagated from the superclass to the subclass. Under multiple inheritance, a subclass may have more than one superclass. The subclass definition then may contain further definitions, which may redefine those in the superclass, a process called *overriding*. Methods and properties may be declared *non-inheritable* and these do not propagate to subclasses. A class may also be an *interface*, consisting only of declarations specifying the types of its properties and methods and possessing no implementation. These serve to separate implementation in the superclass, with subclasses *realising* (implementing) the declarations. *Multiple inheritance* extends (single) inheritance by allowing a class to inherit from more than one superclass, often simplifying the domain model. Particularly, *mixin* classes may be defined — classes which have no superclass but are instead designed to be mixed with other class definitions to produce new combined subclasses. Finally, a *metaclass* is a class from which all classes inherit.

Polymorphism. Polymorphism is the ability of objects belonging to different classes, each subclassing from a class defining a common interface, to respond to a method call of the same name, each one suitable for its own class. In this way, a set of objects of differing class can be grouped, and a call performed on each member of the group which is resolved in a class-specific manner at runtime.

In the context of data mining using inductive logic programming, we concentrate on the use of the object model for closely constraining the class of patterns which may be discovered by the data miner, informed by the

form, structure and characteristics of the data. In this way, the data miner is able to define the domain in a natural and intuitive manner, and furthermore discover patterns in data which exhibits many common characteristics of real-world data, corresponding to elements of the object model. Moreover, existing object modelling techniques may be applied to refine the description of the domain in a form understandable to a domain expert unfamiliar with data mining, producing a shared description which stands alone as an information engineering artefact. Settings in which these characteristics are manifest include those which have a strong notion of individual; learning where objects naturally belong in taxonomies; learning at multiple levels of granularity by selecting the appropriate level of generality in a class hierarchy on which to learn; and finally learning in domains which may be modelled by complex datatypes which are modelled independently of their constituents, for example data which is best expressed in more complex collections than sets or lists. In these situations the semantics of the collection and the learner can be more easily decoupled; learning where exceptions to a general rule and default reasoning are common; learning in the presence of partial knowledge; and learning where domains are naturally simplified by encapsulating and abstracting the interfaces of the objects.

2.2 Declarative bias in ILP

ILP systems in general require a strong declarative bias to be computationally feasible. The first-order-based rich representations that they use to form theories are potentially very expressive. However, in real world domains this expressivity can lead to large search spaces, meaning many ILP tasks are not computationally feasible without restriction of the problem to a simpler but still meaningful task. This restriction is done through the definition of bias. In this section we survey existing approaches to bias in ILP systems, where appropriate comparing them with examples of where bias specific to the object model could be defined.

Bias concerns anything which constrains the search for theories [138]. Bias measures are characterised as belonging to one of three kinds [110]: *language bias*, *search bias* and *validation bias*. The choice of bias is the main point of trade-off between computational efficiency and the quality of theories induced in an ILP system, and requires careful engineering. Tausend [135] shows the effects of common constituents of ILP bias on the resulting hypothesis space, classifying the effects of bias as *syntactic*, largely limiting the size and variable interaction in the produced clauses, and *semantic*, limiting the hypothesis space searched. The role of *metadata* describing the structure and the associated constraints on the construction of the hypothesis space is of critical importance both to the management of overfitting and the space and memory requirements of the data mining task.

Search bias considers the way the system searches through the space of possible theories. Since an exhaustive search of every possible hypothesis is usually infeasible, heuristics are adopted which determine which portions of the space are to be search and which rejected. Validation bias determines the stopping criteria of the learner. Validation bias concerns the quality/running time tradeoff; a learner may stop at the first acceptable theory but may also choose to continue the search for a better (shorter, more comprehensible) theory. Finally, language bias applies constraints on the forms of the clauses in the search space. Since the space of first-order clauses for a non-trivial domain is again infeasibly large, the language bias is defined to arrive at a subset in which to perform a more efficient search. A carefully-designed language bias will effectively reduce the hypothesis space without compromising the search for an interesting clause or theory, straddling the trade-off between efficiency and quality theories. Examples of language bias include only employing Horn clauses, function-free symbols, range-restriction, the moding of predicates and associated output linkage restrictions

(such as all output variables in the head are required to be instantiated by the body, or that a variable may not appear as output more than once, or each output must appear as input to another literal or output in the head) and input linkage restrictions (such as all input variables in the head also appear in the body). Forbidden literals, clauses and conjunctions (integrity constraints) may also be defined. Search parameters introduce bounds on the hypothesis language such as the number of literals or variables in a clause, the maximum depth of a variable and the degree of a clause. Where the bias is defined in terms of a series of syntactic parameters in this way, it is known as *parameterised bias*. Language bias concerns almost entirely the form of the clauses involved in the search and not the process of the search itself.

The incorporation of the object model into ILP thus affects the language bias most significantly. Before considering the forms of bias which the object data model introduces, we review a number of common approaches to bias in existing ILP systems. Unlike in the object data model,the predicate, or relation, is the primary means of organising data, representing both database facts as ground tuples, and intensional background knowledge as Prolog rules. As such, declarations describing characteristics of the data are described on the predicate level and used by an ILP system during clause construction.

Many ILP systems use a basic concept of *type* in order to constrain the constant symbols appearing in clauses or restricting variable unification in clauses to those only of compatible type, thereby restricting variable linkage within the clause. An argument of a predicate is *typed* if symbols appearing in its ground answerset are restricted in this way. The form of typing declarations, and motivation assumptions about type, varies across ILP systems. For example, nominal types consist of a finite set of symbols, whereas continuous types are real-valued. In the FOIL [117] system, nominal types can be defined to be ordered or unordered. Some systems, for example ALEPH [132] and the early MIS [128], incorporate data structures such as lists and binary trees as part of a type declaration. Some ILP systems do not explicitly enumerate types automatically but instead use them purely to restrict sharing of variables, and rely on the user to specify which constants appear in typed arguments. On the other hand, the provision of type information is optional in some systems; TILDE [10] and WARMR [36] will automatically enumerate a type if necessary.

Declarations often associate *modes* to predicate arguments in ILP systems. Although the form of mode declarations vary, typically prefixes to the type declaration declare whether an argument acts as an input variable (symbol +), output variable (symbol -) or is substituted by a constant from the type (symbol #). Systems using mode declarations include ALEPH, PROGOL [107], RSD [84], GOLEM [105] and MIS. More than one mode declaration may exist for a predicate and combination symbols such as +- may be defined in some learners such as WARMR, TILDE and FOIL. The effect of mode declarations is therefore to further restrict the variable sharing possible between arguments; output arguments are substituted with a new variable during clause construction whereas a variable appearing in an input argument is unified with a variable from a previously-occurring variable appearing in an output argument. Constant arguments are substituted in constructed clauses with a value from the type they take[1].

Determinacy and *recall* are two common concepts in many ILP systems. Informally, for an ordered Horn clause $A \leftarrow B_1,...,B_m,B_{m+1},...,B_n$, a term t found in B_{m+1} is *determinate* with respect to B_{m+1} if and only if for every substitution θ such that $A\theta$ and such that $\{B_1,...,B_m\}$ hold, there is a unique atom $B_{m+1}\theta\delta$, with a unique valid ground substitution δ. The *degree* of the literal B_{m+1} is then the number of other terms in B_{m+1}. The *depth* of a literal is 1 if its determinate variable is a function of a variable found in the head of

[1] Instead of a direct mode declaration for a constant term, RSD uses an `instantiate/1` literal, inserted during clause construction, to determine which arguments are occupied by constants.

the clause, and i if it is a function of a set of variables whose minimum depth is $i-1$. Depth and degree together form the notion of ij-determination. From [105], every unit clause is 0j-determinate. An ordered clause $A \leftarrow B_1,...,B_m,B_{m+1},...,B_n$ is ij-determinate if and only if (a) $A \leftarrow B_1,...,B_m$ is (i-1)j-determinate, (b) every literal B_k in $B_{m+1},...,B_n$ contains only determinate terms and has degree at most j. ij-determinacy then expands on determinacy, allowing a precise restriction on the learner's hypothesis language, and defining a class of clauses. ILP systems using concepts of determinacy include ALEPH and GOLEM, in which it was introduced. MIS includes a determinacy flag in the declarations of its predicates and goes further by allowing the specification of total predicates. Where predicates are non-determinate, it is often necessary to bound their non-determinacy for the purposes of clause construction.

The *recall number* is an upper bound on the number of successful calls to a predicate given its input substitutions. For example, in the case of a structured dataset describing trains and their cars, a recall number of 1 on the predicate linking these would determine that a feature constructed with it could address at most one car of a given train [74]. Examples of ILP systems using this form of bias include ALEPH, TILDE, RSD and PROGOL. *Multiplicity* is a related *data relationship* shared by the object model and many ILP learners, such as 1BC, TERTIUS [47] and SINUS. Multiplicity is a natural characteristic relationship in many domains and defines the relationship between the bindings of constants to input variables and the resulting output bindings from successful calls to the predicate. Whereas the recall number limits the number of possible output bindings given a single input binding, a multiplicity specification considers the possible output bindings per single input binding *and* the possible input bindings for a single output binding. Multiplicity implicitly allows the declaration of a predicate's functionality, inverse-functionality and determinacy. A one-to-one multiplicity corresponds to the existence of one output binding per input binding, *i.e.* a determinate predicate. Therefore, more than one call to the predicate is redundant in a constructed clause. A one-to-many multiplicity suggests there is more than one possible output binding per input binding. Finally, a many-to-many multiplicity suggests a more complex relationship — namely that several possible input bindings may lead to the same output binding.

Many other forms of language bias exist. For example, the user can define which predicates follow which others (in the case of WARMR and TILDE), the length of the clause or the number of literals or a bound on the number of variables appearing in a clause. Search bias may also employ statistical and heuristic techniques such as the coverage, accuracy or confirmation of a clause. Language bias need not depend on the a set of individual predicate declarations, but may operate at the clause level. *Clause schemata*, in which template clauses are defined with variables for the predicate symbols. Instantiating these with names yields a family of clauses, thereby defining the language. Cohen [21] proposed a grammatical approach to defining a hypothesis language in GRENDEL, in which a definite clause grammar is used to represent the hypothesis language. Sentences in the language of the grammar are then the valid clauses in the hypothesis language. Nedellec *et al.* [110] proposed a definition of language bias using *clause sets*, defining for a set of possible clauses whether they may or may not appear in a in induced program; *literal sets*, defining a language of clauses by defining template in terms of sets of literals, in which an element of a literal set may be substituted for a conjunction of any of its literals to yield a hypothesis language of possible clauses; and *term sets*, appearing in the literals and specifying a set of terms which may appear in each argument of the literal. These three kinds of set work together to define detailed biases for first-order rules. The DLAB approach [35] to defining declarative bias defines a grammar consisting of a number of predicate templates. A head template and body template are represented by DLAB terms, which in turn consist of either atomic formulae or a formula of the form $min-max:L$ for integers min, max and L, a list of DLAB terms. The language $\mathcal{L}$ is then generated recursively, choosing between *min* and *max* elements

of L to construct a clause. This defines detailed hypotheses accounting for additional clausal properties such as optionality, combined occurrence, and exclusivity of literals. Additionally, determinacy and functionality behaviour may be defined for predicates and their properties exploited to reduce the hypothesis space.

Further forms of bias exist which are of particular relevance to the object data model, defining restrictions to the language which approach those employed in an object representation. Later we will precisely define a scheme for introducting bias by defining metaknowledge for the classes and associations in the domain. An important form of bias in an object data model is inherent in the moding restrictions assumed by methods. In traditional ILP, arguments to predicates may take any mode, whereas in an object model, the object on which a method operates — the *host object* — and its inputs are assumed to be moded as inputs and its return value as an output. The individual-centred representation discussed in section 1.2 which is adopted in a number of learners, provides an example of such a bias. Adopting it places strong restrictions on the variables in constructed clauses — structural predicates provide new variables while property predicates consume them. Adopting such a bias reduces the hypothesis space significantly, and is considerably congruent with the object model.

Object identity is another form of bias employed in a number of ILP systems. It assumes terms in a formula with different symbols are distinct and represent different entities of the domain [127]. For example, under the individual-centred representation, we can ensure that under the object identity bias two components with different variables are distinct. Adopting object identity for clauses whose terms represent object identifiers relates closely to the notion from the object model that each identifier represents a different object in the database. Object identity also simplifies the process of clause construction.

The kinds of type restrictions seen in traditional ILP systems may naturally be extended to reasoning and induction in systems which use type hierarchies as background knowledge. Frisch [51] introduces the idea of a *sort theory*, in which a collection of sorts are defined, each sort representing some subset of individuals in the domain. Accordingly, objects in the database are assigned a sort and variables in induced theories appear with a single distinct sort and sorts are defined to be subsorts of others. By redefining substitution according to the sort theory, clauses can be progressively refined by adapting the sort to which variables are bound. Accordingly, such refinement operates over a smaller hypothesis space than an unsorted counterpart using predicate symbols in a Prolog representation. The refinement is therefore based on substitution, and Frisch argues that incorporating taxonomic background knowledge into such substitions is the most appropriate means of approaching typed refinement. Taking this further, by defining methods at a given point in the inheritance lattice, we can introduce a framework for class-safety, restricting the space of valid clauses. *Default reasoning* is an established logical approach which allows exceptions to be modelled by overriding methods in subclasses. Conventional logic programming systems do not implement such reasoning without use of the `not/1` predicate or a specialised semantics [43] and it is not a consideration specifically taken into account in ILP systems.

In summary, the principal means of introducing bias in ILP systems are via type and mode declarations, determinacy and multiplicity. More specifically, some aspects of the object model may be approached by introducing the individual-centred representation. Specialised logical approaches such as the introduction of a sort theory may further permit object induction in systems taking advantage of them. However, these forms of bias are ill-suited to constraining induction in an object database. Primarily, the type system does not take into account the *hierarchy* of classes nor the definition of methods or properties for subclasses.

2.3 Object logics

The vast majority of ILP systems use Prolog as their resolution engine. The facts and rules of the background knowledge are specified as a logic program, and in many cases the resulting theory is also expressed as a Prolog program. Prolog is an untyped language, and as a result, in many ILP systems it is necessary to rely on the presence of type and mode declarations to enforce type-correctness in the clauses which it constructs. Types are assigned to predicate arguments and constructed clauses such that the type symbols for the arguments in which a given variable appears in a clause must agree. Individual objects are typically typed using facts of the form $\mathtt{t}(\mathtt{x})$ for a type $\mathtt{t}$ and constant $\mathtt{x}$. Variables are only typed as a result of their appearance in arguments of the clause. We consider the applicability of Prolog as well as this types-and-modes bias as applied to the task of object-oriented data mining.

Prolog alone is ill-suited primarily because it is an untyped language. More specifically, unification of terms cannot be determined type-illegal. As a result, there is no means to type-check a predicate call. Resolution can be defined to fail for incompatible types by setting conditions in the body of the rule which check whether the input is of a particular type, and inheritance across these types may be crudely approximated by rules of the form $\mathtt{c}(\mathtt{x}) \leftarrow \mathtt{sc}(\mathtt{c})$ for a subclass $\mathtt{c}$ and superclass $\mathtt{sc}$, but this is cumbersome and ignores the semantics of particular forms of inheritance. Structured terms, which are incompatible with this scheme, frequently appear in Prolog programs, and substructures cannot be typed in this way. The assumptions underlying the types-and-modes bias for induction introduce further unsuitability. The fact that the types must agree in all arguments that a term appears ignores the notion of subclassing. This removes the possibility that the bias can model the process of inheriting a predicate which takes a given type as input to its subtypes. Without knowledge of a type hierarchy, it cannot test the legality of applying a predicate to a term during feature construction. Additional semantics for inheritance and method application assumed under the object model are similarly ill-suited to the types-and-modes bias.

In summary, some basic elements of the object data model may be approximated and imitated within the *deductive* resolution procedure employed by Prolog and the *inductive* methods used in data mining. These include a basic notion of syntactic typing and a crude form of inheritance, and mode declarations for predicates employing these types. However, where more sophisticated semantics for types, inheritance, unification and the legal application of a method are introduced from the object model, Prolog and the inductive biases defined for it become either inconvienient or impossible. Since Prolog is ill-suited to the task of object-oriented induction and deduction, it is therefore necessary to consider an alternative basis for object induction. In order to arrive at a suitable logical framework for our learner, we survey existing logics and logic programming languages in terms of their support of the object model. Later we discuss the related area of description logics.

Just as object databases can be seen as an extension of relational databases, object logics exist which extend first-order logic systems such as Prolog to facilitate deduction in object domains. Object logics and object logic programming systems aim to incorporate principles of object orientation into the representation of data and rules as well as into the resolution procedure. These approaches thus form a logical framework for testing clauses against background knowledge expressed in the object model. Kifer and Wu [67] identify four salient features of the object model: *Complex objects* are objects which are composed from simpler objects. These are typically linked by identifier for objects which provide *object identity*. These together form the basis for the compositional aspect of the object model. *Typing* is the assignment of a class symbol to an object, whereas *inheritance* takes advantage of this assignment to allow methods to propagate to their subclasses and permit

valid unification of terms.

We first discuss systems supporting classed complex objects and then consider issues relating to inheritance. Firstly, the notion of object identity can be thought of as what makes an object individual in a logic or logic programming system. It acts as a handle or identifying term for the object, and is usually modelled by a unique constant. Some logic programming systems such as L&O [93] and LogTalk [100, 99] take the viewpoint that the overall state (facts) of an object represents its identity. Among object logics, the use of an object identifier is far more common.

In traditional logic programming, complex data is is typically represented by complex (untyped) tree-like terms constructed from (possibly nested) functors. The data may also be flattened, in which facts link containing components of a structure to their constituents via newly-introduced constants. Logics based on the object model tend to relate an object identifier to its composite object identifiers via labelled properties. The property then is classed to only permit object identifiers of the appropriate class to appear, in effect typing the structure as a whole. A well-studied approach to typing is to extend the syntax and resolution procedure of Prolog by assigning types to variables and implementing named properties and methods of an object by labelling arguments using functors. The ILP system RHB$^+$ [125] uses such an approach. It did not, however, incorporate the concept of class hierarchy. The logical representation of ψ-terms, proposed by Aït-Kaci et al. in 1986, and the associated languages LOGIN [2] and LIFE [3, 1], extended this framework to include a class hierarchy, and defined a basis for unification in terms of this hierarchy. ψ-terms implemented complex data using nested labelled features. These features could be left unassigned to a value to represent partial information.

In O-logic [90], terms are nested typed object identifiers built up recursively from data values and variables. Each object identifier is associated with a class identifer. An example O-term is staff : john[works $\rightarrow$ dept : cs, salary $\rightarrow$ 20000]. The object john contains two labelled properties defined in its class staff, each of which are classed. These classes form a class structure. john and cs are object identifiers and 20000 is an example of a data value. Variables may appear in the place of any object identifier. A proof procedure is defined for the resulting variablised structures.

The labelled properties in these terms may be scalar- or set-valued, corresponding to multiplicity constraints on associations in the object model. That is, resolution procedures are introduced to reflect whether a property only ever takes one possible value (scalar-valued) or may take a set of possible values (set-valued). There was much interest in the use of set-valued properties, leading to the development of, for example, C-Logic [16], as a first-order logic for complex objects, since it was argued that the incorporation of sets allowed a logic to handle a wider class of structured data. O-logic was later extended [67] to permit reasoning with sets as well as inconsistent information. This was further extended to F-Logic [65], an attempt to fully support object identity, complex objects and inheritance. The revision put much more emphasis on the manipulation of object identifiers, allowing object identifiers representing classed sets to appear as properties, allowing direct unification of set-valued properties in the proof procedure.

The presence of a class hierarchy naturally leads to the notion of *inheritance*. In logic this takes the form of the propagation of property definitions and the availability of method calls. Booch argued that without an inheritance structure, a model cannot be said to be object-oriented at all [11]. Kifer and Wu [66] differentiate two aspects of inheritance. Method declarations are propagated by structural inheritance while their implementations are propagated by behavioural inheritance. The logic programming system L&O incorporates a proof procedure which involves transferring the proof mechanism between different class templates, or theories, where no predicate is available to satisfy a subgoal. By imposing a subclass order on these templates, a

subclass may introduce (augmenting inheritance), override or cancel (differential inheritance, similar to non-inheritable methods). The Prolog++ [98] language introduces a similar inheritance scheme. In ψ-terms, the class-based unification is used to realise inheritance. The proposal of F-logic [66] in 1995 brought about a more comprehensive proposal for inheritance in logic. Strictly, F-logic takes an approach in which *objects* are specialisations of other objects instead of classes being specialisations of each other. Attributes and methods may be declared inheritable or non-inheritable, and the logic supports overriding and polymorphism. With its strong support for complex data as sets, typing and type-correctness, inheritance and encapsulation, F-logic thus implements all the aspects of object-orientation while still possessing a first-order proof procedure.

The object model aims to provide a domain description which separates domain knowledge from implementational knowledge in a domain description which facilitates reuse, analysis, common understanding and interoperability. These aims are shared by ontologies [112], data models common used in knowledge engineering [88] which represent a domain and allow reasoning in it. They are another form of domain description, defining a vocabulary and set of explicit assumptions, usually in first-order logic. Ontologies possess correspondence with the object model. Concepts, representing sets of individuals, either extensionally or intensionally, are are linked to other concepts by a subsumption relationship. They are similar to classes. Slots or attributes define named values possessed by individuals, and link individuals to others via roles. They correspond to properties in the object model. A knowledge base is constructed from terminological and assertional knowledge, the former describing relationships among concepts and axioms (comparable to methods), and the latter membership of individuals to concepts (comparable to class membership assertions). Description logics represent a separate class of logical framework to those already presented. Description logics are a fragment of first-order logic which adopt the ontological model, such that the fragment is as expressive as possible while still being decidable and of desirable computational complexity [134, 7]. Object modelling concepts such as multiplicity [56] and inverse methods are also represented in description logics, though more operational aspects such as method invocation rules and parametric classes are not modelled. Concepts may be negated, conjoined, be subject to value restrictions, employ existential restrictions, requiring a role to map to a given concept [7, 56]. Concept disjunction [126, 56] is also supported. Description logics have been shown to be embeddable into F-Logic knowledge bases as demonstated by Balaban [9], preserving logical implication under a set of semantics for the embedding. Inductive approaches using description logics include work on refinement operators [7], concept formation [41], and hybrid Horn-clause/description logic systems [87]. Other machine learning techniques such as similarity and clustering have also been approached in the framework of description logics [26].

2.4 A language for representing objects

Deduction in object logics can be viewed as a constrained form of deduction in the definite Horn clause representation employed in the majority of ILP systems. These constraints take the form of class membership of terms appearing in a program or query and lead to further restrictions on valid method invocations appearing in the clause. The object logics reviewed in the previous section offer well-founded reasoning procedures for deductive databases expressed in the object model, including complex objects, object identity, method calls, strong typing and inheritance. We consider an approach which *upgrades* these procedures for the purposes of confining the hypothesis space to one more suitable for induction. With regard to this aim, we also wish to enhance the object model with meta-knowledge declarations which aim to place further constraints on a

valid clause, bounding the space of hypotheses being tested and informing search over this space. By choosing appropriate constraints, a strong bias can be defined on this hypothesis space, resulting in a representation which is sufficiently expressive for data mining while still maintaining a hypothesis space of manageable size. The language presented is a subset of F-Logic, enhanced with an extended class-based constraint language and meta-knowledge scheme. F-Logic offers reasoning procedures which take into account the object model, whereas the constraint language and meta-knowledge provide a mechanism to bound the space of hypotheses. We therefore identify two interconnected languages:

- *The object logic language* defines the *relational aspect* of a query. An expression in the object logic language resembles the queries of declarative logic programming languages such as Prolog; the body of the query is an existentially-quantified formula of data expressions — method calls on sets of objects represented by variables, taking input arguments and producing output variables. Executing a query Q against the F-logic database D produces a set of substitutions θ for the variables in the query. θ is then an answer for Q, that is, $D \models Q\theta$. The *relational part* of clauses are expressed in the object logic language. In this thesis we may also refer to the data expressions as *literals*, by analogy with logic programming.

- *The constraint language* primarily serves to constrain the set of substitutions θ during the reasoning process by which an answerset is generated. This is principally performed by assigning a class to each term in Q, using a language of class constraints, as well as means to impose cardinality and class constraints on the substitutions for variables in Q. The *constraint part* of clauses are expressed in the constraint language.

The final ingredient concerns the construction of the queries during induction. The set of induced queries Q can be significantly reduced by the use of *metaknowledge* about the methods used in data expressions and the structure of the classes used in the constraint language.

The resulting language is named CORLOG and is used to represent the examples, background knowledge, queries and induced theories. Both sublanguages have associated reasoning procedures, which are combined in order to solve a constrained goal. Inductive processes are adapted to take advantage of the constraint and meta-knowledge schemes in order to search the hypothesis space appropriately. We consider these three elements of CORLOG in detail later in chapter 3. Before doing so, we first review the basic syntax and semantics of F-Logic with a view to adopting a restriction of it and enhancing that restriction for the purposes of induction. Sections 2.4.1, 2.4.2 and 2.4.3 summarise relevant aspects of the F-Logic framework from [66]. Section 2.4.4 gives an example domain and section 2.4.5 compares the semantics of F-Logic with that of logic programming. For clarity, we set F-Logic expressions in a sans-serif typeface throughout this thesis.

2.4.1 Basic syntax: the alphabet and well-formed formulae

Before proceeding with the presentation of F-Logic, it is important to note a number of deviations and related notational changes which were made from it. Firstly, the logical framework and learner presenting in this thesis assumes that all methods are inheritable and set-valued. For clarity, we adapt the notation from the original F-Logic. These changes are indicated in the text.

Definition 2.1 (F-Logic alphabet [66]). The alphabet of an F-logic language L consists of the following symbols:

- A set of object constructors F, acting as function symbols of the logic and used to represent attributes, methods, class and object identifiers. Symbols of zero arity form the constants while symbols of arity ≥ 1 may be used to construct compound terms out of simpler ones.

- An infinite, enumerable set of variables V.

- Symbols ( ) [ ] ; , $\rightarrow$ $\Rightarrow$ [2] and logical connectives and quantifiers: $\vee \wedge \neg \leftarrow \forall \exists$

Id-terms are the equivalent of terms from first-order logic. The ground id-terms play the part of (logical) object identifiers (also known as *oids*) and make up the Herbrand Universe. Methods, their arguments, classes in the system as well as the identifiers for objects all share the same space of id-terms. Classes are therefore reified in F-Logic.

Definition 2.2 (id-term [66]). An id-term is a first-order term formed composed of function symbols and variables as in predicate calculus, *i.e.*:

- Any constant is a term (with no free variables).

- Any variable from V is a term (whose only free variable is itself).

- Any expression $\mathsf{f}(\mathsf{t}_1,...,\mathsf{t}_n)$ of $n \geq 1$ arguments (where each argument t_i is a term and f is a function symbol from F) is a term. Its free variables are the free variables of any of the terms t_i.

- Nothing else is a term.

By convention, constants are represented by lowercase symbols, *e.g.* c and variables are represented by uppercase symbols, *e.g.* V. Observe that complex terms ($\mathsf{f}(\mathsf{t}_1,...,\mathsf{t}_n)$ above) are included, but are never adopted in this thesis except in the definition of parametric classes. The simplest kind of formulae in F-logic are called *molecular F-formulae* or simply *molecules*. Well-formed formulae are built up from simpler formulae using the connectives $\vee$, $\wedge$, $\neg$ and quantifiers $\forall$, $\exists$.

Definition 2.3 (molecular F-formulae, molecule [66]). A *molecular F-formula* or *molecule* is one of the following:

- An *is-a assertion* of the form C :: D or of the form O : C, where C, D and O are id-terms.

- An *object molecule* or *data expression* of the form $\mathsf{O}[\mathsf{M}_1;...;\mathsf{M}_n]$ where each M_i is a method expression. A method expression can be either a *non-inheritable data expression*, an *inheritable data expression* or a *signature expression*.

We define the data expressions and signature expressions later. However, we introduce the term *host object* for the O appearing in an object molecule. Informally, this denotes the object the method is being called on. The expression $\mathsf{O}[\mathsf{M}_1;...;\mathsf{M}_n]$ is shorthand for the expression $\mathsf{O}[\mathsf{M}_1] \wedge ... \wedge \mathsf{O}[\mathsf{M}_n]$. Complex F-formulae are constructed from simpler F-formulae following conventions from first-order logic, *i.e.* that all molecular formulae are F-formulae and if ϕ and ψ are F-formulae and X a variable, then so are $(\phi \vee \psi)$, $(\phi \wedge \psi)$, $\neg\phi$, $\forall \mathsf{X}\psi$ and $\exists \mathsf{X}\phi$. Nothing else is a F-formula.

[2] In the notation used in [66], a large number of possible arrow symbols were used. In our simplification, we use only those shown here.

A *clause* in F-Logic, as in Prolog, is an implication of the form $\phi \leftarrow \psi$. The left-hand side ϕ is known as the head of the clause, and the right-hand side ψ the body of the clause. The body ψ consists of a conjunction of object molecules (*cf.* literals), intended to be an existentially-quantified conjunctive formula. As in conventional logic, this implication is equivalent to $\phi \vee \neg\psi$, and is also called a rule. A *definite clause* is a clause in which ϕ consists of exactly one positive F-molecule. A clause in which a method symbol from the head also appears in the body is a *recursive* clause. A *query* in F-Logic is a clause with an empty head. A *fact* in F-logic is a clause with an empty body. A *program* is a set of clauses, and constitutes the database. Predicates in Prolog may be considered *extensional* or *intensional*. Extensional predicates consist of ground facts, each one corresponding to one instance of a relation in a database, or alternatively, a record. Intensional predicates are expressed as rules, defining the set of facts belonging in a relation as a rule, which then models each of the ground facts. These definitions carry over to the F-logic case with an analogous rule syntax as described above. Again, as in Prolog, clauses and F-molecules containing no variables are called *ground*.

2.4.2 Object molecules and method expressions

The object logic component of a F-Logic clause consists of a conjunction of object molecules, typed by a series of method signature expressions. The syntax of F-Logic and its implementation FLORA-2 differs significantly. For the sake of clarity we adopt a notation closer to the implementational form.

Definition 2.4 (Method expression, adapted from [66]). An (inheritable) *method expression* is one of the form $M(I_1, I_2, ..., I_n) \rightarrow R$[3]. All of M, I_i and R are id-terms. M is a method symbol[4] and I_i and R are arguments. n is termed the *arity* of M and $n \geq 0$.

Both represent a method invocation called on its host object O as in definition 2.3. Method invocations are typed by signature expressions, bounding the classes of objects that a method's host object, inputs and outputs may range over, a key difference from Prolog.

Definition 2.5 (Signature expression, adapted from [66]). A *signature expression* is an expression of the form:

$$C[M(V_1, V_2, ..., V_n) \Rightarrow A] \tag{2.1}$$

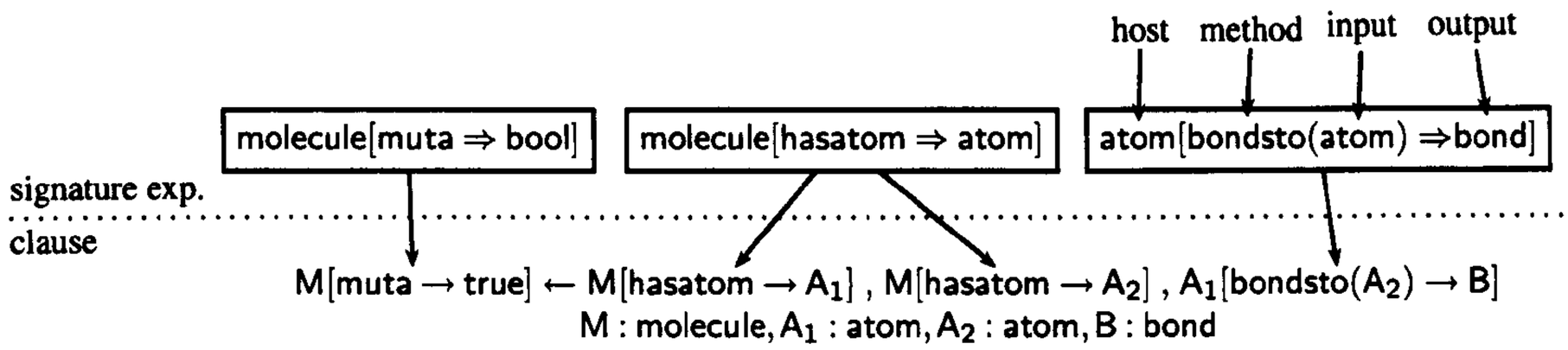
C, M, V_i and A are id-terms representing classes. A is the class of the result returned by the method M when invoked on an object of class C with arguments of class V_i.

For induction, we assume a strict moding over the arguments in the molecule. In method expressions, each I_i is an input (as well as the host object O) and each R is an output. We deviate from standard F-logic by considering all method expressions to be inherently set-valued. That is, the method call may succeed many times, giving multiple possible results for R. A *property* is a method for which $n = 0$, *i.e.* of the form $M \rightarrow R$. The moding scheme adopted thus assumes a one-way nature to methods, relying on input terms which are already assumed to be bound to a finite number of constants, and introducing a single new term (variable or constant).

---

[3] Non-inheritable expressions were signified by the arrow notation $\bullet\!\!\rightarrow$ and inheritable ones by the arrow notation $\rightarrow$ in [66]. Since we do not consider non-inheritable methods in this thesis, we simplify the notation to simply $\rightarrow$ for both. Furthermore, since all methods are assumed to be set-valued, we use only $\rightarrow$ rather than the double-headed notation adopted in [66]

[4] Owing to the higher-order syntax of F-Logic, strictly speaking, M can be a variable. However, in this work, we never regard it as such.



Signature atoms (above) cover method expressions appearing in clauses. Each signature atom defines the method symbol and the types of its host, input and output arguments. The clause consists of a relational part (top line) and a constraint part (bottom line).

Figure 2.1: Coverage of a data expression by signature expressions

Figure 2.1 illustrates the relationship between data expressions appearing in clauses and their covering signature expressions. Each data expression therefore is intrinsically linked with a signature expression. We say that the signature expression *covers* the data expression and that the data expression *is covered by*, or *matches*, the signature expression.

Methods are therefore associated, using method signatures, with a class C of objects on which they are invoked. Suppose that a particular object o is defined to be of class C' in the database. A non-inheritable method expression is only applicable to this object if $C = C'$, whereas an inheritable method expression may be applied if C is a superclass of C'. An inheritable method expression defined in a class C may be redefined in a subclass C'. This redefinition is the basis for overriding inheritance behaviour during deduction. When a class is called on an object, this structure brings about the need for a method selection rule, since two entirely different behaviours may be defined depending on the class of the host object. We therefore require the object logic to select the method defined in the most specific class possible.

Example 2.6 (Method selection rule). Consider three classes A, B and C. A is a superclass of B and B of C. Method M has definitions in classes A and C. According to this selection rule, a call to M will result in the definition in class A if called on an object of class A or B, and the definition in class C if called on an object of class C.

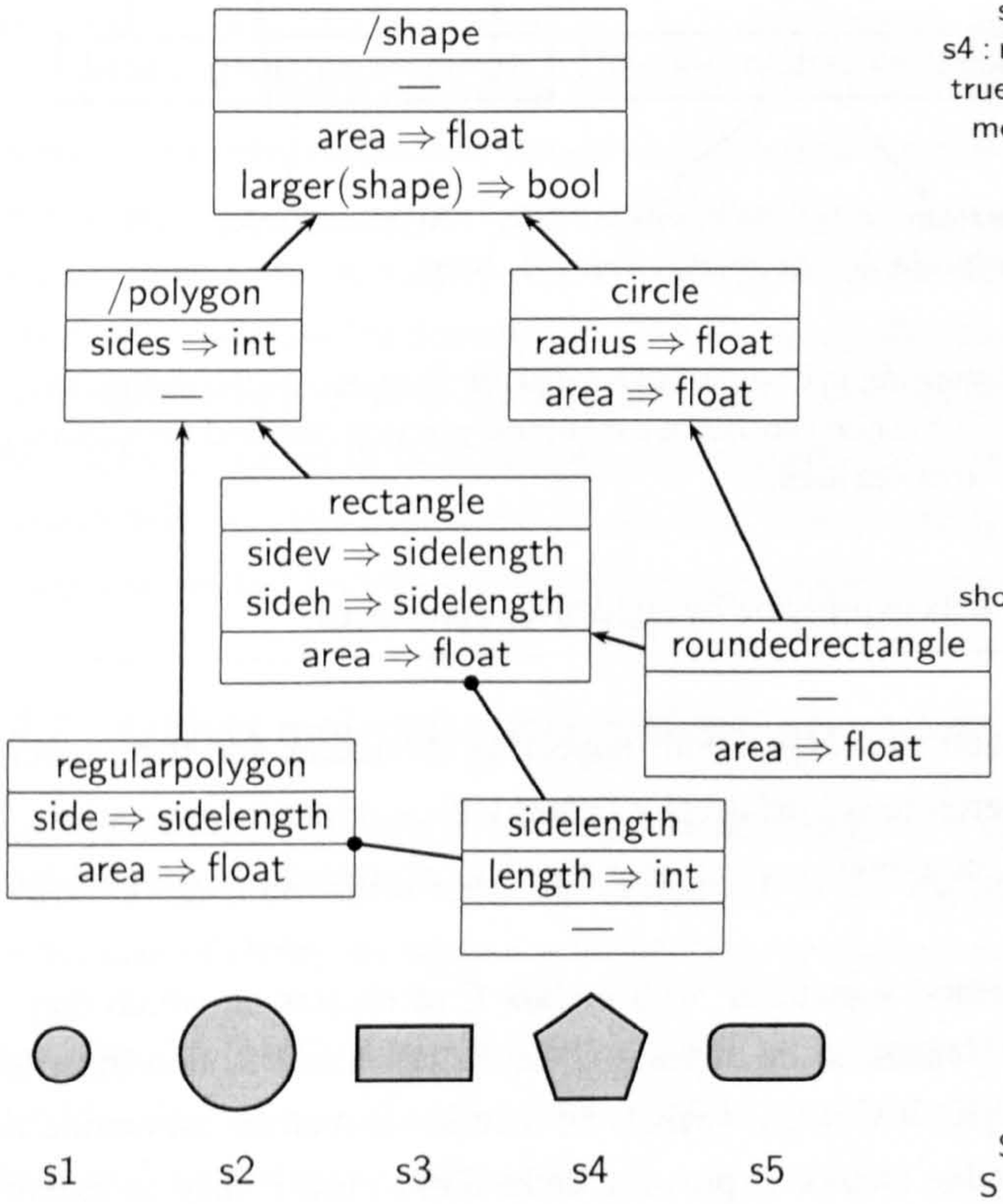
2.4.3 Defining classes: is-a molecules

Terms in F-Logic may be bound to classes. Most object-oriented viewpoints see a class as a template to which an object conforms, specifying its data and methods. In F-Logic, a class, denoted by an id-term, also denotes the set of objects belonging to that class. Strictly, an individual object is modelled as a class containing one element.

Definition 2.7 (class membership molecule). A *class membership* molecule is an expression of the form O : C, where O and C are id-terms. This asserts that O is a member of class C.

The inheritance hierarchy is defined in terms of set of subclass definitions.

Definition 2.8 (subclass definition molecule). A *subclass definition molecule* is an expression of the form



Class membership:
s1 : circle. s2 : circle. s3 : rectangle.
s4 : regularpolygon. s5 : roundedrectangle.
true : bool. false : bool. short : sidelength.
medium : sidelength. long : sidelength.
Subclass relationships:
polygon :: shape. circle :: shape.
rectangle :: polygon.
regularpolygon :: polygon.
roundedrectangle :: rectangle.
roundedrectangle :: circle.
Data expressions:
s1[radius → 1]. s2[radius → 2].
s3[sidev → 1]. s3[sideh → 2].
s4[side → 1]. s4[sides → 5].
s5[sidev → 1]. s5[sideh → 2].
short[length → 1]. medium[length → 2].
long[length → 3].
Signature expressions:
circle[radius ⇒ float].
polygon[sides ⇒ int].
rectangle[sideh ⇒ sidelength].
rectangle[sidev ⇒ sidelength].
shape[area ⇒ float].
shape[larger(shape) ⇒ bool].
sidelength[length ⇒ int].
Method definitions:
S1 : shape[larger(S2) → X] ← ...
S : circle[area → X] ← ...
S : rectangle[area → X] ← ...
S : regularpolygon[area → X] ← ...
S : roundedrectangle[area → X] ← ...

Part of a class hierarchy describing shapes. Each box represents a class, with the name in the top section and middle and bottom sections listing signatures for the properties and methods respectively. Pointed arrows are drawn from the subclass to the superclass and a line shows composition, with a circle on the end of the containing class. Below are five example objects and to the right, the domain expressed in F-Logic.

Figure 2.2: An example shapes domain

C :: D where C and D are id-terms. This asserts that C is a subclass of D - that is, where the object of class C (resp. D) are denoted O_C (resp. O_D), then $O_D \subseteq O_C$.

2.4.4 An example domain

A simple domain is shown in figure 2.2. The individuals s1 to s5 are some example shape objects expressed according to the domain. The diagram focuses on the inheritance structure, in which roundedrectangle is an example of multiple inheritance, since it combines aspects of circles and rectangles. The method area is overridden in several classes. It is defined in shape as an interface, *i.e.* it has no implementation. When we refine the object model later, we will see that classes may be *abstract*. Abstract classes are used as a means of abstracting common elements in two classes or class structures, but which do not contain objects themselves. Calls to the method on objects of class shape would therefore fail, but since shape is declared abstract (with

prefix / in the class diagram), no object is defined to be of class shape. The domain also defines the values of enumerable types, for example a shape may have a sidelength of value short, medium or large (1, 2 and 3 units respectively). Datatypes such as float and int represent the floating-point and integer numbers. The signature definitions capture the class structure, and method definitions, replaced by ellipsis for brevity, implement the methods. The compositional aspect of the data exists in this domain as the property link between, for example, rectangle and the lengths of its sides (sidev for vertical and sideh for horizontal).

2.4.5 Semantics and reasoning in F-Logic

Resolving queries in object logic is very similar to resolving queries in traditional logic programming languages such as Prolog. We therefore present the semantics of F-Logic from [66] in the context of conventional logic programming, considering first the form of a substitution. A *substitution* θ in F-Logic is a mapping from the variables in a formula to id-terms. They may be represented by a finite set of the form $\theta = \{V_1/t_1, ..., V_n/t_n\}$, for $V_i \neq V_j$ where $i \neq j$. The *instance* $C\theta$ resulting from applying a substitution θ to a clause C is the result of simultaneously replacing each occurrence of each V_i with its t_i. Where each t_i is a constant id-term, the substitution is called a *ground substitution*. Substitutions may be applied to F-molecules by distributing them over the molecules' components, including the method symbol. Then, $(O : C)\theta = O\theta : C\theta$, $(C :: SC)\theta = C\theta :: SC\theta$ and $O[M(A_1, ..., A_n) \rightarrow R])\theta = O\theta[M\theta(A_1\theta, ..., A_n\theta) \rightarrow R\theta]$. Substitutions extend to F-formulae by distributing them over logical connectives and quantifiers in a similar way.

The semantics of a clausal relational reasoning system are typically defined in terms of two aspects of reasoning. The model-theoretic semantics define the set of ground facts which are logical consequences of a program, allow us to define logical entailment in a logical language and assign meaning to sentences in a logic. The proof-theoretic semantics, on the other hand, concerns the process of performing inference in clausal logic, and allows us to decide whether a formula entails (is a logical consequence of) another.

The model theory of F-Logic, in common with many clausal relational reasoning systems, follows from the model theory of logic programming, defined in terms of a program P and a first-order language L, and in particular, the Herbrand interpretation.

Definition 2.9 (Herbrand universe, Herbrand base [111], Herbrand interpretation [43]). The *Herbrand universe* U_L for L is the set of all ground terms which can be formed out of the constants and function symbols appearing in L. The *Herbrand base* B_L for L is the set of all ground atoms which can be formed out of the predicate symbols[5] in L and the terms in the Herbrand universe U_L. A *Herbrand interpretation* I of L is a mapping from the Herbrand base into the set of truth values $\{true, false\}$.

The Herbrand base B_L is thus partitioned into the set of ground atoms $P(t_1, ..., t_n)$ such that $I(P(t_1, ..., t_n)) = true$ in one subset and $I(P(t_1, ..., t_n)) = false$ in the other, usually abbreviated to include only the atoms mapping to $true$.

Definition 2.10 (Satisfaction by a Herbrand interpretation). We say a positive literal l is satisfied by I if $l \in I$ and a negative literal $\neg l$ is satisfied by I if $l \notin I$. A ground clause $l_1 \vee ... \vee l_n$ is satisfied by I if at least one l_i is satisfied by I. A clause C is satisfied by I if and only if all ground instances of C are satisfied by I.

[5]The predicate symbols include both the intensional and extensional predicates.

The meaning of a logic program is usually defined by its Herbrand model, defined in terms of its satisfaction by the interpretation. F-Logic uses a structure to define the Herbrand interpretation. The general notion of a structure is defined as follows:

Definition 2.11 (Structure). A structure S consists of: (i) A set A, the universe of S; (ii) For each constant symbol, an element $c^A \in A$; (iii) For each n-ary function symbol f, a function $f^A : A^n \rightarrow A$. (iv) For each n-ary relation symbol R, a subset $R^A \subseteq A^n$. If A is the Herbrand universe of a language L, S is the Herbrand structure of L.

Before proceeding, an important aspect of F-Logic which should not be overlooked is the handling of equality. F-Logic maintains an equality predicate to denote that two terms refer to the same object. This arises from two sources; cyclicity in the subclass hierarchy and multiple definitions of single-valued method calls. For example, two *single-valued* expressions (in the original notation) alice[father $\rightarrow$ bob] and alice[father $\rightarrow$ robert] would imply that bob and robert are the same object, written bob $\doteq$ robert. Similarly, the assertions lorry :: truck and truck :: lorry mean that truck and lorry refer to the same class. Equality is defined to be reflective, symmetric and transitive and may be used in substituting terms during deduction, known as paramodulation, and in factoring $U(\mathcal{F})$ into equivalence classes of $\doteq$. However, since we do not consider single-valued methods and assume acyclity of the class hierarchy in the approach presented in this thesis, equalities of this sort do not arise. We therefore omit it from the logical framework presented.

This definition is refined to F-structures for the object model of F-Logic. Note that since we do not consider single-valued methods or non-inheritable methods nor consider Prolog-style predicates of the form $p(a_1,...,a_n)$, the semantics presented here are a simplification of those given in [66].

Definition 2.12 (F-structure [66]). If $\mathcal{F}$ is the set of function symbols in a language L, an F-structure is a tuple $I = \langle U, \prec_U, \in_U, I_{\mathcal{F}}, I_{\rightarrow}, I_{\Rightarrow} \rangle$. U, is the domain of I [6]. $\prec_U$ is an irreflexive partial order on U defining the subclass relationship and $\in_U$, a binary class-membership relation, determined by is-a assertions. $I_{\mathcal{F}}$ is a total mapping from id-terms to object ids. The set U is a set of all actual objects within a possible world I. Ground id-terms are used to represent logical object ids. Each k-ary object constructor $f \in \mathcal{F}$ is interpreted by a function $U^k \rightarrow U$. $I_{\rightarrow}$ (for non-inheritable methods) is a partial function[7], giving the mapping of data expressions. Method calls are therefore represented by partial functions, and may be overloaded by arity. $I_{\Rightarrow}$ are partial functions giving the mappings for types, *i.e.* specifying the type of a method, and incorporating the semantics of polymorphic functional types. Summarising Kifer and Wu [66], this mapping, for a method M, is characterised as a set of tuples (of any length) taking class symbols from U, describing the classes for which the values in an i-ary method call to M may legally take. The tuples take values of a partial anti-monotonic function from U^{i+i} (the classes of the arguments) to $P_{\uparrow}(U)$ an upward-closed subset of U. Anti-monotonicity in this sense permits an input argument to be replaced by one of its subclasses. If $\prec_U$ is extended to operate over sequences of classes, $\rho : U^k \rightarrow P_{\uparrow}(U)$ is anti-monotonic if for *sequences* of class symbols $u, v \in U^k$ such that $v \preceq_U u$, if $\rho(u)$ is defined, then $\rho(v)$ is defined and $\rho(v) \supseteq \rho(u)$. Upward closure accounts for the fact that an output value of a given class v are also substitutable by its superclasses. More formally, a set $V \subseteq U$ is upward closed if $v \in V$, $v \prec_U v'$ for $v \in U$ implies $v' \in V$.

An F-Logic formula may be satisfied by an F-structure. We present satisfaction properties of F-structures deriving from the model theory above and the framework presented in [66].

[6] More strictly, $U(\mathcal{F})/=$, the quotient of $U(\mathcal{F})$ induced by equalities in the language.

[7] The functions are of arity $k+1$, the first argument referring to the host object and the remainder to the k arguments.

- The is-a relationship defines a partial order over $U(F)$. (i) Reflexivity: $I \models \mathsf{p} :: \mathsf{p}$ (ii) Transitivity: If $I \models \mathsf{p} :: \mathsf{q}$ and $I \models \mathsf{q} :: \mathsf{r}$ then $I \models \mathsf{p} :: \mathsf{r}$. (iii) Acyclicity: If $I \models \mathsf{p} :: \mathsf{q}$ and $I \models \mathsf{q} :: \mathsf{p}$ then $I \models \mathsf{p} \doteq \mathsf{q}$. These establish that the subclass relation is a partial order over the id-terms $U(\mathcal{F})$.

- Subclass inclusion: If $I \models \mathsf{p} : \mathsf{q}$ and $I \models \mathsf{q} :: \mathsf{r}$ then $I \models \mathsf{p} : \mathsf{r}$. This establishes that an object is a member of its class' superclasses.

- Type inheritance (where permitted by the interheriability of the method) If $I \models \mathsf{p}[\mathsf{m}(\mathsf{q}_1, ..., \mathsf{q}_k) \Rightarrow \mathsf{s}]$ and $I \models \mathsf{r} :: \mathsf{p}$ then $I \models \mathsf{r}[\mathsf{m}(\mathsf{q}_1, ..., \mathsf{q}_k) \Rightarrow \mathsf{s}]$. This permits propagation of structure from a class to its subclasses.

- Input and output restrictions: $I \models \mathsf{p}[\mathsf{m}(\mathsf{q}_1, ..., \mathsf{q}_i, ..., \mathsf{q}_k) \Rightarrow \mathsf{s}]$ and $I \models \mathsf{q}'_i :: \mathsf{q}_i$ then $I \models \mathsf{p}[\mathsf{m}(\mathsf{q}_1, ..., \mathsf{q}'_i, ..., \mathsf{q}_k) \Rightarrow \mathsf{s}]$. This states that if a method accepts a class q_i as input, it will also accept any of its subclasses q'_i. $I \models \mathsf{p}[\mathsf{m}(\mathsf{q}_1, ..., \mathsf{q}_k) \Rightarrow \mathsf{r}]$ and $I \models \mathsf{r} :: \mathsf{s}$ then $I \models \mathsf{p}[\mathsf{m}(\mathsf{q}_1, ..., \mathsf{q}'_i, ..., \mathsf{q}_k) \Rightarrow \mathsf{s}]$. This states that an output value of class r is also of any superclass s of r.

With respect to logical entailment, an F-structure I is defined to be a model of a closed formula ψ if and only if $I \models \psi$ according to the above satisfaction rules. It follows from this that if Σ is a set of formulae and ϕ is a formula, $\Sigma \models \phi$ if and only if ϕ is true in every model of Σ. In F-Logic, the answer to a query Q is the set of all ground instances of Q which are logically implied by the program (database) P. In traditional first-order logic programming, an *answer* to a query Q is a substitution θ for the variables in Q. The answer is valid with respect to the program P if $P \models Q\theta$. The *answer set* is the set of answers to Q that are valid and which ground Q ($Q\theta$ is ground). F-Logic defines the answer set as:

Definition 2.13 (F-Logic answer set [66]). The set of answers to a query Q with respect to an F-program P is the smallest set of molecules that (i) contains all instances of Q that are found in the canonical model of P and (ii) is closed under $\models$.

Canonical models arise from the presence of single-valued methods mapping to distinct values, requiring the model to be equality-restricted. We do not consider single-valued methods, however. In F-Logic, type-correctness is considered part of the meta-theory of the logic. In order to define it, it is necessary to define the notion of a typed H-structure.

Definition 2.14 (H-structure [66]). Given an F-structure for a set of clauses S, the corresponding H-structure is the set of ground molecules that are true in the F-structure. An H-structure I is typed if (i) every (inheritable) data atom in I is covered by some signature in I; (ii) if an (inheritable) data atom $\mathsf{c}[\mathsf{m}(\mathsf{a}_1, ..., \mathsf{a}_k) \rightarrow \mathsf{v}] \in I$ is covered by a signature of the form $\mathsf{d}[\mathsf{m}(\mathsf{b}_1, ..., \mathsf{b}_n) \Rightarrow \mathsf{w}] \in I$, then $\mathsf{v} : \mathsf{w} \in I$.

The conditions for type-correctness are defined as follows:

Definition 2.15 (Type-correctness in F-Logic, type coverage [66]). Let I be a structure and α a data atom of the form $\mathsf{c}[\mathsf{m}(\mathsf{a}_1, ..., \mathsf{a}_k) \rightarrow \mathsf{v}] \in \mathsf{I}$ and β is a signature atom of the form $\mathsf{d}[\mathsf{m}(\mathsf{b}_1, ..., \mathsf{b}_k) \Rightarrow ...]$. We say β *covers* α if for each $i = 1, ..., k$ we have $\mathsf{c} :: \mathsf{d}, \mathsf{a}_i : \mathsf{b}_i \in I$. I is a typed H-structure with respect to data expressions if the following conditions hold: (i) every data atom in I is covered by a signature atom in I (ii) If a data atom $\mathsf{c}[\mathsf{m}(\mathsf{a}_1, ..., \mathsf{a}_k) \rightarrow \mathsf{v}] \in I$ is covered by a signature of the form $\mathsf{d}[\mathsf{m}(\mathsf{b}_1, ..., \mathsf{b}_m) \Rightarrow \mathsf{w}] \in I$, then $\mathsf{v} : \mathsf{w}$.

The proof theory of a logical language concerns the process of performing inference. Again, we start from a logic programming setting and refine the notions into F-Logic. Deductive inference is the process of deciding whether a formula F logically entails another formula G, written $F \models G$. Given an initial set of clauses P (sentences, axioms), a system applying inference uses a set of inference rules R to *derive*, or deduce, new sentences by repeated application of the inference rules in R. The notation $P \vdash_R C$ denotes that the clause C can be derived from P using R (where the context of R is clear, $P \vdash C$ may be used). In order to be computationally tractable, the rules in R usually manipulate clauses in a purely syntactic manner, capturing the properties of implication from the model theory. The primary rule in logic programming systems, including those that manipulate objects, is resolution [120]. $P \vdash C$ is therefore an approximation to $P \models C$, and in comparing them, we identify the properties of soundness and completeness.

Definition 2.16 (soundness and completeness). For a set of inference rules R and its resulting inference relationship $\vdash_R$, a program P and a clause C: R is *sound* if for all P and C, if $P \vdash_R C$ then $P \models C$; R is *complete* if for all P and C, if $P \models C$ then $P \vdash_R C$; R is *refutation complete* if for all P and C, if $P \models C$, then $P \wedge \neg C \vdash_R \square$, where $\square$ is the empty clause.

Refutation-completeness is a more meaningful property in assessing inference procedures, since resolution is known to be incomplete for even propositional logic [43]. The inference procedure of F-Logic is refutation complete. Like most relational logics, F-logic inference uses the notion of a unifier. We present the general definition first.

Definition 2.17 (unifier, most general unifier). A substitution θ is a *unifier* for two id-terms, is-a molecules, or P-molecules T_1 and T_2, if $T_1\theta = T_2\theta$. A unifier $\theta = mgu(T_1, T_2)$ is a *most general unifier* if for every unifier μ of T_1 and T_2, there exists a substitution σ such that $\mu = \sigma\theta$.

Unifying substitutions also provide a basis for generality in induction, to be introduced in chapter 4. Of particular interest in this regard are most general unifiers. These notions carry over into F-Logic, where syntactic conventions introduce an alternative concept of unification. This reflects the fact that unification at the object level is done between objects sharing the same object identifier and done simultaneously on all attributes and methods of that object, possibly expressed as sets. For example, we may attempt to unify $L = \mathsf{john}[\mathsf{age} \rightarrow 31]$ and $L' = \mathsf{john}[\mathsf{age} \rightarrow \mathsf{X}, \mathsf{children} = \{\mathsf{alice}, \mathsf{bob}\}]$, in which L' contains more than one attribute/method and has a number of solutions represented as a set[8]. Accordingly, unifiers do not require identity, but require that a molecule L_1 is mapped into a submolecule of another molecule L_2. Under the assumption that attributes/methods are grouped together, unifiers then act asymmetrically under F-Logic, θ being a unifier of L_1 *into* L_2 if and only if $L_1\theta \sqsubseteq L_2\theta$, where $\sqsubseteq$ represents the subset relation over molecules in L_1 and L_2. Similar definitions follow for tuples of terms. In this thesis, we do not assume molecules are syntactically grouped by object identifier in this way and therefore do not consider this asymmetry. Additionally, in F-Logic, a single most general unifier does not always exist. More formally, and following the previous definition, we consider a set of most general unifiers.

Definition 2.18 (more general unifier, most general unifier, complete set of most general unifiers). A unifier α is *more general* than a unifier β, denoted $\alpha \trianglelefteq \beta$ if there exists a substitution θ such that $\beta = \theta\alpha$. α is

[8] An expression with a set valued variable may unify with any single member of that set. For example, unifying $\mathsf{a}[\mathsf{set} \rightarrow \mathsf{X}]$ and $\mathsf{a}[\mathsf{set} \rightarrow \{\mathsf{b}, \mathsf{c}\}]$ may yield unifiers $\{X/b\}$ or $\{X/c\}$.

then *most general* if $\forall\beta, \beta \trianglelefteq \alpha \rightarrow \alpha \trianglelefteq \beta$. A set of unifiers Σ is *complete* if for every unifier θ there is an $\alpha \in \Sigma$ such that $\alpha \trianglelefteq \theta$.

Returning to conventional logic programming, two core deduction rules are employed — resolution and factoring — often expressed as one. Resolution of two clauses $l \leftarrow b_1, ..., b_n$ and $h \leftarrow c_1, ..., c_{i-1}, l, c_i, ..., c_m$ infers a *resolvent* $h \leftarrow c_1, ..., c_{i-1}, b_1, ..., b_n, c_i, ..., c_m$. More formally,

Definition 2.19 (resolution, binary resolvent, complementary pair, standardised apart). *Resolution* is the process of applying the rule $\{(\psi \vee \phi), (\neg\psi \vee \chi)\} \vdash \phi \vee \chi$ for formulae ψ, ϕ and χ. $\phi \vee \chi$ is said to be the *binary resolvent* of $\psi \vee \phi$ and $\neg\psi \vee \chi$ and formulae ψ and $\neg\psi$ are said to form a *complementary pair*. It is necessary to rename variables in each of the clauses being resolved upon in order to obtain the most general resolvent. The clauses are then said to be *standardised apart*.

In practice, ψ, ϕ and χ are usually clauses. If $C_1 = L_1 \vee ... \vee L_i \vee ... \vee L_m$ and $C_2 = M_1 \vee ... \vee M_j \vee ... \vee M_n$ are two clauses which have been standardised apart, the clause

$$(L1 \vee ... \vee L_{i-1} \vee L_{i+1} \vee ... \vee L_m \vee M_1 \vee ... \vee M_{j-1} \vee L_{j+1} \vee ...M_n)\theta$$

is the binary resolvent for a $\theta = mgu(\{L_i, \neg M_j\})$. $L_i\theta$ and $M_j\theta$ are then the complimentary pair. This approach to resolution is not sufficient for deduction in that it does not allow derivations of resolvents containing fewer literals than those in the two clauses being resolved. In order to reduce the number of disjuncts produced in a clause, *factoring* must be applied. Factoring operates on a single clause C, containing a nonempty set $\{L_1, ..., L_n\}$ of literals unifiable via an mgu θ. The clause C' obtained by deleting the unified literals $\{L_2\theta, ..., L_n\theta\}$ from $C\theta$ is a *factor* of C. Approaches to deduction combine the processes of binary resolution and factoring in the following way; a *resolvent* of two *parent clauses* C_1 and C_2 is a binary resolvent of a factor of C_1 and a factor of C_2. The literals resolved upon (those substituted by the mgu) are those unified by the factors. Referring to the repeated application of resolution rules described in the introduction, a *derivation* of a clause C from a set of clauses Σ is then a sequence of clauses such that each clause is a member of Σ or a resolvent of two earlier clauses in the sequence.

Reasoning in an object-oriented paradigm, however, requires extensions to capture the properties of types and is-a relationships. As such, additional inference rules are added, modelling these relationships. The inference rules of F-Logic are defined across twelve rules and one axiom. There are three core inference rules; resolution, factoring and paramodulation. Several rules, including paramodulation, exist as a result of equalities inferred from single-valued methods. We consider first the core inference rules. Resolution and factoring are defined similarly in F-Logic to logic programming, the most notable difference being the asymmetric mgu. The *resolution* rule states that if $W = \neg L \vee C$, $W' = L' \vee C'$, and $\theta = mgu(L, L')$, then from W and W' we derive $(C \vee C')\theta$. The *factoring* rule states that if $W = L \vee L' \vee C$ for positive literals L and L', and $\theta = mgu(L, L')$, then $(L \vee C)\theta$. If $W = \neg L \vee \neg L' \vee C$ for negative literals $\neg L$ and $\neg L'$, and $\theta = mgu(L, L')$, then from W and W' we derive $(\neg L' \vee C)\theta$.

The remaining rules concern the derivation of clauses involving is-a assertions and the type restrictions introduced by signature atoms. The *is-a reflexivity* rule states that $\forall X, X :: X$. The *is-a transitivity* rule accounts for the transitive nature of the subclass hierarchy. If $W = (P :: Q) \vee C$, $W' = (Q' :: R') \vee C'$ and $\theta = mgu(Q, Q')$, then from W and W' we derive $(P :: R' \vee C \vee C')\theta$. The *subclass inclusion* rule accounts for the fact that each object is also a member of its class' superclass. If $W = P : Q \vee C$, $W' = Q' :: R' \vee C'$ and

$\theta = mgu(Q, Q')$, $(\mathsf{P} : \mathsf{R}' \vee C \vee C')\theta$. The *type inheritance* rule allows subclasses to inherit a signature from a superclass. If $W = \mathsf{P}[\mathsf{M}(\mathsf{I}_1, ..., \mathsf{I}_n) \Rightarrow \mathsf{T}] \vee C$, $W' = (\mathsf{S}' :: \mathsf{P}') \vee C'$ and $\theta = mgu(P, P')$ then from W and W' we derive $\mathsf{S}'[\mathsf{M}(\mathsf{I}_1, ..., \mathsf{I}_n) \Rightarrow \mathsf{T}] \vee C \vee C'$. The *input restriction* rule introduces a class requirement on inputs to methods. If $W = \mathsf{P}[\mathsf{M}(\mathsf{I}_1, ..., \mathsf{I}_i, ..., \mathsf{I}_n) \Rightarrow \mathsf{T}] \vee C$, $W' = (\mathsf{I}_i'' :: \mathsf{I}_i') \vee C'$ and $\theta = mgu(I_i', I_i'')$ then from W and W' we derive $W = \mathsf{P}[\mathsf{M}(\mathsf{I}_1, ..., \mathsf{I}_i'', ..., \mathsf{I}_n) \Rightarrow \mathsf{T}] \vee C$. Finally, the *output restriction* rule introduces a similar requirement on the output of a method. If $W = \mathsf{P}[\mathsf{M}(\mathsf{I}_1, ..., \mathsf{I}_n) \Rightarrow \mathsf{R}] \vee C$, $W' = (\mathsf{R}' :: \mathsf{R}'') \vee C'$ and $\theta = mgu(R, R')$ then from W and W' derive $\mathsf{P}[\mathsf{M}(\mathsf{I}_1, ..., \mathsf{I}_n) \Rightarrow \mathsf{R}''] \vee C \vee C'$. A merging rule combines information in different object molecules into a single object molecule by forming the union of its constituent atoms. A further rule removes tautological molecules of the form $\mathsf{P}[]$. The last rule covers the handling of equality, and enables terms to be found equal to be substituted for each other, known as paramodulation. Recall that we do not consider single-valued methods and assume acyclity of the class hierarchy in the approach presented in this thesis. As a result, the rules do not apply in our deduction framework.

2.5 Conclusion

In this thesis we adopt inductive logic programming as a means of performing multi-relational data mining using object-oriented data. Appropriate to the logical setting of ILP, such data is expressed in a deductive database, and regularities in it in a logical language. In this chapter we have discussed the form of such a logical language as a basis for the logical framework used in the remainder of this thesis.

We began by considering the general object model and the characteristics which the system must possess in order to be considered object-oriented. The characteristics of object, class, properties and methods, inheritance and polymorphism. In the context of ILP, object orientation may be primarily considered a restriction in bias. Accordingly, we considered the role of bias in reducing an ILP system's search space, reviewing the various methods by which ILP systems introduce bias into their languages and search, concentrating on those elements which are of particular relevance to the object model, namely the types, modes and cardinality declarations made in many ILP learners. The key elements of the object representation were also identified, namely the host object, the individual-centred representation and the use of object identity. Proposed forms of bias relevant to the existence of a sort and inheritance hierarchy were considered.

Many of these forms of bias assume the use of Prolog as a representation language. We examined some of the shortcomings of Prolog and these choices of bias for modelling object data and the rules that are used to find regularities in the data. The need for a special-purpose object logic was identified. Accordingly, a review of object and description logics is presented, considering how they support the elements of object-orientation. One such logic, F-Logic, forms the basis for the logic framework we adopt for learning. We presented its syntax, semantics and reasoning mechanisms. In chapter 3, we adapt it for inductive logic programming.

Chapter 3

CORLOG: A logical language for object-oriented induction

It can be seen from the definition of F-Logic in the previous chapter that the object model lends itself to a constrained form of first-order deduction. The notion of class and its association with arguments in a clause via signature molecules restricts the terms which appear in them, defining which constants or object identifiers may appear or restricting useful variable bindings between arguments of compatible class. There is therefore a strong notion of type-correctness, implemented in the flexible type system of classes. Particuarly, signature methods declare which methods may be applied to which terms and strict mode declarations further serve to constrain variable sharing. Such restrictions are clearly of use in deduction, and also provide a role in introducing a bias for induction. In this chapter, we further define the role of the object model for induction, and propose extensions to constrain the language further in the context of existing principles of the object model. The resulting language is called CORLOG.

The chapter is structured as follows. Section 3.1 describes the form of clauses and features in CORLOG. CORLOG clauses consist of a relational part — described in section 3.2 — and a constraint part — described in section 3.3. The domain description, and in particular the role of metaknowledge, is discussed in section 3.4. Section 3.5 concludes.

3.1 Clauses and features in CORLOG

We view F-Logic clauses in terms of separate components, adopting these components for the purposes of defining extended concepts in CORLOG. Particularly, we view an F-Logic clause as possessing two separate components. Firstly, it has a *relational part*, constructed from data expressions, and expressing the relationships between terms in the expressions. Secondly, variables in the relational component are constrained to belong to classes in a *constraint part*. Separate to the clause are associated method signatures which define the type-validity of the clause via rules derived from the semantics presented in the preceding chapter. CORLOG extends the constraint component to represent a more flexible, but still class-based, constraint system. Moreover, the signature methods can be viewed as a form of metaknowledge, defining the language bias in terms of the classes. CORLOG extends the notion of metaknowledge to other aspects of the data model not represented in F-Logic

but of use in restricting the language bias for induction. In this thesis, we argue that by adopting the object model and enhancing it with extensions to the class and metaknowledge benefits the inductive process. These benefits include reduction of the hypothesis space by identifying impossible and invalid assignments of terms to arguments and the introduction of new forms of redundancy and impossibility in clauses, enabling them to be eliminated from the hypothesis space. The object model thus serves as a means of defining a hypothesis language over which induction is performed, via a domain model expressed in the object model.

Before considering the individual aspects of this extended object view, we first state some important definitions, continuing from the basic F-Logic definitions introduced in section 2.4.

Definition 3.1 (CORLOG clause and related definitions). A CORLOG clause C is an implication of the form

$$C = l_0 \leftarrow l_1 \wedge ... \wedge l_n \wedge c_1 \wedge ... \wedge c_m \tag{3.1}$$

where each l_i $(1 \leq i \leq n,\ n \geq 0)$ is a CORLOG *literal* consisting of an F-logic data expression, and each c_i $(0 \leq j \leq m,\ m \geq 0)$ is a constraint expression (in F-Logic, a class membership constraint). l_0 is a positive CORLOG literal comprising the *head* (or conclusion or consequent) of the clause, whereas the conjunction of literals $l_1, ..., l_n, c_1, ..., c_m$ comprise the *body* (or condition or antecedent). The literals l_i $(1 \leq i \leq n, n \geq 0)$ are termed the *relational part* of the clause and the constraints c_j $(1 \leq j \leq m, m \geq 0)$ are termed the *constraint part* of the clause. A clause such that $n = 0$, i.e. a clause with an empty body, is termed a *fact*. Logically, a CORLOG *query* is a CORLOG clause with an empty head, representing information which is never true, *i.e.* a refutation of the body against the database.

By analogy with traditional ILP, CORLOG clauses are the representation for the examples, background knowledge, queries and the induced theories. In this thesis, we adopt the approach and terminology of propositionalisation for the induction, described in much more depth in chapter 5. Accordingly, we define the notion of a CORLOG feature, a specific type of clause.

Definition 3.2 (CORLOG feature). A CORLOG feature f_i is an implication (and CORLOG clause) of the form

$$\mathsf{O[f_i(I_1, ..., I_n) \rightarrow R]} \leftarrow l_1 \wedge l_2 \wedge ... \wedge l_n \wedge c_1 \wedge c_2 \wedge ... \wedge c_m \tag{3.2}$$

where the meaning of l_j $(1 \leq j \leq n)$ and c_k $(1 \leq k \leq m)$ is the same as in definition 3.1. The classes of the head variables O, I_i and R define the interface of the feature. Each must be constrained with a class constraint in $c_1 \wedge ... \wedge c_m$.

We will occasionally omit the arguments and use the notation f_i to refer to a feature, where the context is clear. We make the distinction between a feature and a clause as a result of the intended application to propositionalisation. In propositionalisation, features are the building blocks of induced classification rules; the bodies of induced rules consist of conjunctions of features. The head variables of the classification rule r are themselves necessarily class-constrained and must be compatible with the constraints in each feature f_i as well as take the same mode. Specifically, each host object or input variable v of class c appearing in a feature f_i must also appear as an input variable in the head of r, constrained to be of class c or a subclass of c. Likewise, each output variable v of class c appearing in a feature f_i must also appear as an input variable in the head of r, constrained to be of class c or a superclass of c.

As well as requiring linkage of the variable as O and R between the head and the body (*i.e.* that O appears as an input to one of the literals in the body of the clause and R an output from one of the literals in the body), we further specialise the notion of a feature by imposing a number of syntactic restrictions on it. For the purposes of induction, we consider the class of CORLOG features which consist of definite Horn clauses, *i.e.* those with exactly one positive (head) literal, where each literal is a method expression. Furthermore, we require that the clause is function-free. Structured terms involving functors are of limited use in CORLOG since terms are related to subterms through facts involving method expressions. In a sense, CORLOG programs may be said to be intrinsically flattened [122, 81]. Structured data in the object database is necessarily represented by method expressions linking objects to their constituent parts, which play the role of the functors typically used in Prolog structured terms. Functors may appear only in a parameterised form of a class, discussed in section 3.3.1.

We will later see that the induction technique adopted is based on a form of θ-subsumption. In order to overcome problems involving recursion, as well as general issues involving evaluation of recursive clauses, we assume that the language contains no recursive clauses. The choice to disregard recursive clauses results from the emphasis on the induction task in this thesis as classification of complex data structures. The individual-centred representation applied to the object model assumes a class of individual which possesses the class to be predicted as a named property. In this setting, the induction of recursive clauses is only necessary where the class label of an individual is dependent on the class label of its constituent objects. We consider this requirement to be a special case and do not consider it for this thesis. Furthermore, the learning of recursive clauses is not amenable to propositionalisation [81], the ILP technique adopted in this thesis, since the transformation into propositional form may result in much unnecessary complexity, not least considering execution time for the recursive query during transformation.

We also require that each variable in the feature is constrained to belong to some class in the system. A valid CORLOG feature must therefore include in its list of constraints c_i a membership assertion for each variable. We term this a *class-constrained feature*, discussed in more depth later. In terms of search efficiency, we also wish features to be undecomposable, *i.e.* that there is no feature f such that the literals in the body of f can be partitioned into disjoint conjunctions $f_1, ..., f_n$ such that no pair (f_i, f_j) share non-head variables. Equivalently, this means that f cannot be re-expressed as the conjunction of two or more other valid clauses in $\mathcal{L}$. Again, we discuss this later. Finally, we wish to ensure non-redundancy of each literal in a feature, *i.e.* that no literal (method expression) in f can be removed from the body of f without changing the set of valid substitutions for the variables in f. Equivalently, no literal in f is implied by the presence of any other.

Returning to the notion of compatibility of a set of features for a given class-constrained target relation, we identify the notion of a feature set — a set of features which may appear in a conjunction to form the body of a classification rule.

Definition 3.3 (CORLOG **feature set).** For a feature f_i, denote its interface — the classes to which its head variables are constrained — as $int_i = \langle O_i, \{I_{i_1}, ..., I_{i_n}\}, R_i \rangle$. A set of features $f_1, ..., f_m$ forms a CORLOG feature set FS with an interface $int_{FS} = \langle O_{FS}, \{I_{FS_1}, ..., I_{FS_n}\}, R_{FS} \rangle$ if every O_i is constrained to be of class O_{FS} or a superclass of O_{FS}, every I_{i_j} is of class I_{FS_j} or a superclass of I_{FS_j} and every R_i is of class R_{FS} or a subclass of R_{FS}.

The interface thus places a contract of type-correctness on each feature in the feature set. Each feature therefore necessarily uses the head variable O as an input to at least one of its constituent literals, and furthermore is used in a type-legal way. Constructing features is of particular importance in propositionalisation, the

approach to ILP taken in this thesis. In our approach, we restrict the form of a CORLOG feature to one which always has a constant in its output argument and takes no inputs other than the host object. We term this type of feature a *simple* CORLOG *feature*. Under this restriction, the CORLOG feature adopts the following form:

Definition 3.4 (Simple CORLOG feature). A simple CORLOG feature is a feature of the form

$$\mathsf{O}[\mathsf{f}_i \rightarrow \mathsf{r}] = l_1 \wedge \ldots \wedge l_n \wedge c_1 \wedge \ldots \wedge c_m \tag{3.3}$$

where the meaning of l_j $(1 \leq j \leq n)$ and c_k $(1 \leq k \leq m)$ and O is the same as in definition 3.2.

The feature is necessarily Boolean-valued. We consider for the remainder of this chapter the general form of a CORLOG feature and specialise it to the simple case in chapter 5.

Having divided a clause into its relational part and constraint part, we characterise an object-oriented knowledge base in CORLOG by dividing it into a disjoint set of knowledge declarations as follows. We sometimes use the term *database*, where the context is clear, to refer to a knowledge base.

Definition 3.5 (CORLOG knowledge base). A CORLOG knowledge base is a tuple $KB = \langle D, C, S, M \rangle$, where:

- $D = D_P \cup D_M$: *Database facts and methods.* D_P comprises a set of ground CORLOG facts defining properties of objects in the database, *e.g.* bob[child → alice] in which bob and alice are object identifiers. Similarly, D_M defines the results of method invocations in terms of deductive rules.

- C: *Class membership assertions.* Objects are also defined by the classes they belong to via class membership assertions (*e.g.* alice : person).

- S: *Subclass and subsumption definitions.* Declarations defining the subsumption of classes by others, and in particular, defining where one class is a subclass of another, as in tractor :: vehicle.

- $M = M_S \cup M_M$: *Method signatures and metaknowledge.* Method signatures in M_S assign classes to the host, input and output arguments of methods and properties, enforcing typing restrictions on facts in F_D and D, for example person[child ⇒ person]. Supplementary metaknowledge, in M_M, further describes semantic relationships in data expressions and across the class hierarchy. The forms of this metaknowledge is discussed later in this chapter.

We consider each of these elements of the knowledge base in turn, considering the role of each and the manner in which it is extended from F-Logic into CORLOG. Firstly, we consider the relational part of a feature and in particular interactions among literals for example in terms of variable sharing and decomposability properties of features. We then consider forms of class constraints applicable to terms in a feature, its coverage and consistency with other constraints, and the effect of classing on feature validity via signatures and the interaction of input and output arguments, particularly with respect to valid variable sharing and unification. Finally, we discuss other forms of metaknowledge, both those which relate to the relational part of a feature and those which relate to the constraint part of a feature, arriving at a set of useful metaknowledge declarations.

3.2 The relational part of a feature

The syntax and semantics for the relational part of a feature are adopted closely from our restriction of F-Logic. Reviewing the forms of this restriction, we assume no scalar methods — all methods are assumed to be set-valued, *i.e.* in general methods may succeed many times. We prefer to capture this special behaviour in the

metaknowledge rather than the logic. Furthermore, we wish to avoid complications with equality maintanence and maintain correspondence with logic programming, which does not make this distinction.

The propositionalisation process works by solving a query in which the variables appearing in host object (and input) arguments to a CORLOG feature are necessarily bound to some constant representing the individual and object identifiers associated with it. Similarly, the output arguments resulting from a CORLOG feature must necessarily succeed for a finite set of constants in order for them to appear in the transformed attribute/value table. Placing these restrictions on CORLOG clauses introduces a necessary bias over the language of valid clauses, namely linkedness. Since method arguments in CORLOG are typed and moded, we ensure these requirements hold by requiring that where an input variable appears in the clause, it has been instantiated by previous literals. That is, assuming a left-to-right approach to clause evaluation, there are a finite number of ground substitutions for a variable V appearing as an input. Likewise, method expressions are expected to introduce a finite number of possible substituions in their output arguments. This is a special case of linkedness [54] from traditional ILP, in which at least one variable appearing in a literal is linked, possibly through others, to the variables in the head literal. All variables appearing in the head of the clause thus appear in the body. In ILP this notion goes by the various names of *generative*, *range-restricted* or *connected*. The converse is not true; not all variables appearing in the body appear in the head, however, and the clause is thus not necessarily *constrained*. Because of this assumption that method expressions assume the presence of ground substitutions of inputs and the introduction of ground substitutions for outputs, we impose several conditions on the syntactic structure of a clause, with respect to its variable sharing.

Definition 3.6 (Linked CORLOG feature). A CORLOG feature $F = \mathsf{O}[\mathsf{f_i}(\mathsf{I_1},...,\mathsf{I_n}) \rightarrow \mathsf{R}] \leftarrow \mathsf{I_1},...,\mathsf{I_n},\mathsf{c_1},...,\mathsf{c_m}$ associated with a set of moded head variables H from $\{\mathsf{O},\mathsf{I_1},...,\mathsf{I_n},\mathsf{R}\}$, is *linked* if: (i) All output variables in H appear in at least one output argument in the body of F; (ii) The *host object* input variable in H appear as host object input variables in the body of F[1]; (iii) Each variable appearing in an input argument of a literal in the body of F must appear as the variable appearing in an output argument in the body of F, or as an input argument from H. The variable is then said to be *consumed* by the input;

Observe that for simple CORLOG features, it is sufficient that the host object variable in the head of F appears as a host object variable in the body. The notion of *decomposability* extends the notion of linkage by considering the linkage properties of a feature. This work adopts the technique of propositionalisation for induction. In such a setting, if a feature in a feature set can be expressed as a conjunction of two features, it is redundant, and furthermore increases the dimensionality of the data and reduces search efficiency.

More formally, we define a decomposable feature as follows:

Definition 3.7 (decomposable feature). A *decomposable feature* is a feature $F = \mathsf{O}[\mathsf{f_i}(\mathsf{I_1},...,\mathsf{I_n}) \rightarrow \mathsf{R}] \leftarrow \mathsf{I_1},...,\mathsf{I_n},\mathsf{c_1},...,\mathsf{c_m}$ with a relational part consisting of literals $L = \{l_1,...,l_n\}$, associated with a set of moded head variables H from $\{\mathsf{O},\mathsf{I_1},...,\mathsf{I_n},\mathsf{R}\}$, which can be partitioned into disjoint subsets of literals $\{L_1,...,L_m\}$, $L = \cup_i L_i$, such that there exist no pair of literals (l_i, l_j), $l_i \in L$, $l_j \in L$, $i \neq j$, which share variables not in H.

Where variables are shared between arguments in literals in a clause, they may be viewed as variable dependencies. The creation of decomposable features naturally comes about as a result of substitutions breaking variable dependencies between literals in a feature. These dependencies can be visualised as a graph between literals in a clause. We term this graph the *variable dependency graph* $VD(C)$ of a clause C. The graph is defined as follows:

[1] In order to preserve the individual-centredness of the clause, non-host inputs are considered optional for linkage in the body.

Definition 3.8 (variable dependency graph). A graph $VD(C)$ is the variable dependency graph of a clause C if:

- Each node represents a literal — either the head literal or a body literal. These nodes are labelled l_i, where l_i refers to the ith literal in the body and l_0 the head literal.

- Each output/input dependency between argument is represented by a directed edge with a single arrowhead between the nodes representing the literals where these variables appear. The edge goes from a literal in which the variable appears as an output to one which it appears as an input, thus representing a variable provider/consumer relationship in the clause. Input/input relationships and output/output relationship between the head literal and a body literal are also represented in this way, *i.e.* by a line with no arrowhead at either end.

- If the variable labelling an edge is a host variable, the edge is denoted by a double-headed arrow. This shows that it is important for the linkage resulting from the individual-centred representation adopted, and should not be substituted.

- Variables appearing either as the host object of the head literal or a variable appearing as the output object of the head literal are denoted by underlining the variable labelling the edge. In order to be consumed, and for the clause to be linked, these variables must either appear as input arguments in the body literals if they are host objects in the head literal, or appear as output arguments in the body literals if they are output arguments in the head literal.

- Constants (ground instances) do not appear on the graph.

This visualisation of variable sharing both among body literals and between the head and body literal then allows us to analyse decomposability properties of features and determine their validity with respect to the object model.

We consider some examples of how decomposability comes about and illustrate the process on their variable dependency graphs.

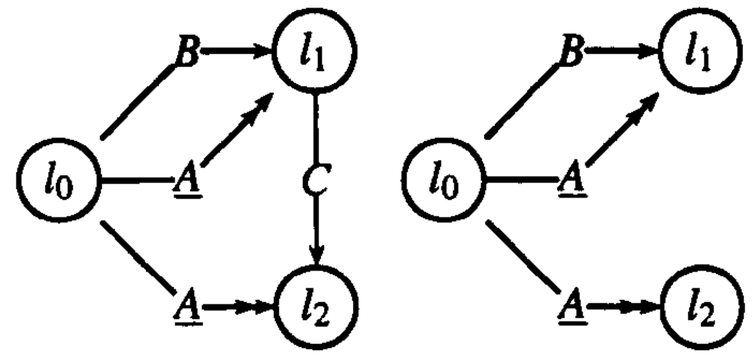
Example 3.9 (Decomposability of CORLOG features). Consider the CORLOG feature C_1 below, assuming arbitrary class constraints on the variables but such that the constants c and d belong to the class constraining C and D respectively. Also consider a substitution $\theta = \{C/c, D/d\}$. Then,

$$C_1 = A[m_0(B) \rightarrow E] \leftarrow A[m_1(B) \rightarrow C], A[m_2(B) \rightarrow D], A[m_3(C) \rightarrow E], A[m_4(D) \rightarrow E]. \quad (3.4)$$

$$C_1\theta = A[m_0(B) \rightarrow E] \leftarrow A[m_1(B) \rightarrow c], A[m_2(B) \rightarrow d], A[m_3(c) \rightarrow E], A[m_4(d) \rightarrow E]. \quad (3.5)$$

The feature $C_1\theta$ is now decomposable; new features can be constructed with subsets of its literals. For example, if l_i is the literal containing the method m_i, $C_1\theta$ can be decomposed into new legal clauses $C_{1_1} = l_0 \leftarrow l_1, l_3$, $C_{1_2} = l_0 \leftarrow l_1, l_4$, $C_{1_3} = l_0 \leftarrow l_2, l_3$ and $C_{1_4} = l_0 \leftarrow l_2, l_4$.

Example 3.10. Let us consider another example, in which the output of the head is a constant, as in the



C_2 $C_2\theta$

Figure 3.1: Two examples of variable dependency graphs

following feature. Consider another substitution $\theta = \{C/c, D/d\}$ producing the substituted feature $C_2\theta$.

$$C_2 = A[m_0(B) \rightarrow e] \leftarrow A[m_1(B) \rightarrow C], A[m_2(C) \rightarrow d]. \quad (3.6)$$

$$C_2\theta = A[m_0(B) \rightarrow e] \leftarrow A[m_1(B) \rightarrow c], A[m_2(c) \rightarrow d]. \quad (3.7)$$

l_1 and l_2 are now unlinked and $C\theta$ can be decomposed into new legal features $C_{2_1} = l_0 \leftarrow l_1$ and $C_{2_2} = l_0 \leftarrow l_2$.

Figure 3.1 shows the graphs of features C_2 and $C_2\theta$. Consider $VD_B(C)$, the portion of the graph containing only the nodes representing body literals, *i.e.* the nodes L_B for $L_B = \{l_i | 1 \leq i \leq n\}$. From figure 3.1, we can readily see that C_2 is non-decomposable, since $VD_B(C_2)$ is connected. $C_2\theta$, on the other hand, is decomposable, since $\{l_1\}$ and $\{l_2\}$ form disconnected subgraphs in $VD_B(C_2\theta)$. However, each decomposed feature is valid, since each is linked via A from l_0. The feature is decomposable where the graph $VD_B(C)$ is disconnected. Each connected subgraph then represents a subclause consisting of a set of literals represented by its nodes. Each of these subclauses either is valid, *i.e.* produces a linked subfeature with the head, or does not and is invalid.

In the graph interpretation, if the host input variables from the head literal appears on a link *to* at least one literal in the subgraph, and the output variable in the head appears in a link from at least one literal in the subgraph, the subclause is valid. These links are those represented by double-headed arrows. Otherwise, it is invalid. We can see that applying a substitution to a clause can affect its decomposability properties, turning a non-decomposable clause into a decomposable one. Later, we determine the set of valid substitutions for a clause — those that do not result in it becoming decomposable. We revisit this framework to determine the set of valid substitutions which may be made during refinement of a clause by considering properties of its variable dependency graph.

3.3 The constraint part of a feature

Constraints can be characterised as expressions which place restrictions on the valid assignments of values to variables of a feature. The principal means of restriction is by associating classes with variables, specifying that the variable may only be assigned values which belong to the given class. A class is then viewed as a collection of values. This constraint is expressed as an is-a molecule of the form x : c, where c is an existing class in the database. An example of a simple class constraint is x : integer for a class integer. posinteger, representing the set of positive integers, may then be defined a subclass of integer. The constraint x : posinteger is then necessarily stricter. More than one class membership assertion may apply to a term. In this case, separate

constraints $O : C_1, ..., O : C_n$ are reformulated as a *conjunction of classes* $O : (C_1, ..., C_n)$. O is a member of $(C_1, ..., C_n)$ where O is a member of every C_i, $1 \leq i \leq n$. More formally, a variable V with a class membership constraint $V : C_V$ for $C_V = C_1, ..., C_n$, V may only be bound to a value v if v is defined to be of each of the classes C_i, $1 \leq i \leq n$ in the knowledge base. During resolution, the set of valid ground substitutions for V are determined in the same way. Similar results follow for variable substitution.

Two fundamental operations exist in the handling of constraints. Firstly, in order for the class structure to be used to specify a restriction on the terms, it is necessary to be able to check whether the term is a member of that class. We refer to this operation as an *instance check*, and say that the term is an *instance* of the class. Secondly, since induction is the technique of interest, we are interested in establishing a structure of *subsumption* over constraints, where c subsumes c' (c' is at least as strict as c) if every individual in the database which is an instance of c' is also an instance of c. This subsumption relationship thus structures the space of constraints, and forms the basis for making features more specific. The class hierarchy directly determines the subsumption hierarchy of constraints — where sc :: c, the constraint t : c subsumes the constraint t : sc. We define subclass among conjunctive classes as follows:

Definition 3.11 (subclassing among conjunctive classes). A conjunctive class c' is a subclass of another conjunctive class c where, considering c and c' as sets, each element of $e \in c$ has at least one element $e' \in c'$ such that e' is a subclass of e or $e = e'$. Necessarily, c' represents a stricter condition. c'' is *reduced* from c' where all simple classes which are a superclass of another in c' are removed.

Classes which are superclass of others in a conjunctive class are redundant, and the reduced form is therefore equivalent to the non-reduced form in terms of constraint. We consider a class constraint to be a conjunction of classes in this work, which includes the special case where they are a single class. Accordingly, we will sometimes discuss a class as being a single class or a conjunction of classes, depending on context.

Each variable in a CORLOG feature (including head variables) is constrained by one or more class membership assertions. More generally, id-terms are also implicitly constrained by their membership of classes in the knowledge-base. Every valid CORLOG feature is said to be *class-constrained* in this way.

Definition 3.12 (class-constrained feature). A feature F consisting of method expressions F_d in its relational part and class membership molecules F_m in its constraint part is a *class-constrained feature* if, for every variable V in appearing in F_d, there exists in F_m a class membership constraint $V : C$ where C is a conjunction of valid classes in the database.

3.3.1 The role of parametric classes

Having considered the conjunction of classes, we return to the notion of a parametric class, introduced in section 3.3.1. Parametric classes offer a more detailed form of constraint, expressed as functors. There are two parts of a parametric class specification $P\langle C_1, ..., C_n\rangle$; the P part is referred to as the parametric class, and $C_1, ..., C_n$, its parameters. Classes with no parameters are called *simple*, and those with parameters *parametric*. In our setting, we restrict the classes which are taken as parameters to be simple classes.

Parametric classes as applied to learning allow the modelling of arbitrary collection classes, which take general and class-restricted parameters, the latter imposing requirements by specifying a superclass on the parameters taken. Such collection classes include standard data structures such as sets, lists, trees and graphs. By inheriting a class *graph⟨atom⟩* from *molecule*, a molecule is abstracted as a group of atoms, implemented

by the class *graph*, which requires methods provided by the class *atom* — other parametric classes such as one representing a list may not place any such requirements on its parameters. The graph-like nature is thus abstracted by the parametric class. Alternatively, parametric classes may be explicitly defined in the database. The parametric class therefore appears as a typing construct and a means of introducing new method invocation validity on an existing class by abstracting it. By modelling a collection with a simple class, no class restriction is specified on the contained object, and therefore no method invocations can be assumed by the parametric class. In parametric classes, this problem is overcome. The bound class serves to type the methods defined in the parametric class. During induction, this typing is of even more importance.

The methods in a parametric class often rely on the availability of methods in the class parameters they take. For example, consider a class representing a list of words $list\langle word\rangle$. Instead of maintaining an internal list of object identifiers, it may rely on each word implementing a property which links to its next and previous words (word[next ⇒ word], word[prev ⇒ word]) in order to access them as a list. In object-oriented programming, the arguments of parametric classes can be required to implement a given class in order to capture this method dependency. A class acts as a *bound* on a parameter, establishing a contract, or guarantee of existing methods, on the parameters.

Parametric class specifications with bounds appear in two parts of the knowledge base. Firstly, they may be specified in method signatures as an extension of the class which the method takes in its arguments. Secondly, they may appear in subclassing declarations, either to abstract structure from an existing class or to denote membership to a restricted domain. Methods specific to trees are then added to those in *graph* to form the *tree* class, as usual. In this thesis we are interested in mining data expressed in, and abstracted into, common structures. In the object model, parametric classes implement these abstract structures. We clarify this idea with a fuller example:

Example 3.13 (Abstracting structure in a parametric class). Consider a molecular domain in which we want to view a molecule as a graph of atoms. We may make the declaration molecule :: graph(atom). Notionally, this asserts that the class molecule is connected to a set of individuals of class atom, which can be manipulated in a graph context. With regard to implementation, a database which might realise this manipulation might be one in which graph expects its parameter to implement methods linksto (representing edges) and hasnode (representing nodes), in order to fulfil methods on the resulting graph, and atom does this. Such a contract could also be realised by an abstract class linked, a superclass of atom acting as an interface class, in which the method linksto is defined. molecule is not a parametric class, but still inherits method calls from graph(atom) allowing methods to be called on the set of constituent atoms as a graph (for example averagedegree for the average degree of the nodes). In summary, graph has method hasnode, so molecule must implement (or inherit it from graph), and requires linksto to exist in its parameter, so atom must implement it.

3.3.2 Type-safe features in the presence of class constraints

Despite the considerable difference in syntax, we may view an object logic such as CORLOG or F-Logic as being a restriction of conventional logic programming, the type restrictions arising from an additional set of conditions described in the proof and model theories. The introduction of constraints on terms in an F-feature relative using a system of inheriting class constraints, together with the notion of type-correctness arising from method signatures, defined from F-Logic in definition 2.15, brings about further restrictions. These restrictions are manifested in the sharing of variables among the arguments of methods in literals, as well as the id-terms

which may appear in these arguments.

During induction, variable sharing among different arguments of literals in a feature arises as a result of unification substitutions, and the appearance of ground id-terms in arguments appears as a result of instantiating substitutions. The set of valid substitutions is discussed in more depth in chapter 4. Here we summarise the rules from the definition of F-Logic which define the valid interactions between variables in the relational part of a clause and the constraints in the constraint part.

There are a number of elements of the object model which affect type-safety. Firstly, with respect to the class membership constraints c_i on each variable, we only permit unification among terms of comparable class. Secondly, we consider the classes of the arguments in method signatures which cover literals appearing in the feature. The method signatures impose a contract on the types of variables which may appear in the arguments of a matching method, leading to constraints on type-correctness as well as input/output linkage. Finally, with regard to specific metaknowledge defined in the domain, variables must be unified in such a way that the resulting clause does not violate the conditions introduced by metaknowledge. That is, the clause should be metaknowledge-respecting.

Following from the properties of F-structures, the appearance of a variable in a feature places requirements (from arguments which consume variables) and guarantees (from arguments which consume variables) on the range of classes which ground substitutions for the variables may take. These guarantees and requirements interact in order to determine the type-safety of a variable with respect to its involvement in a feature. These aspects of a method's types — its interface — are often referred to as the contract of the class.

Definition 3.14 (Conditions for type-correctness of a variable in a class-constrained feature). Consider a variable V appearing in a class-constrained feature $F = l_0 \leftarrow l_1,...,l_n,c_1,...,c_m$ consisting of a relational part $L = \{l_0,...,l_n\}$ and constraints $C = \{c_1,...,c_m\}$, adopting the following definitions of consumer and provider arguments. We say V is a *consumer* if it appears in an input argument in the body of the feature or as an output argument in the head of the feature. We say V is a *provider* if it appears in an output argument in the body of the feature or as an input argument in the head of the feature. V is type-correct if:

- For each consumer a where V appears in L, with corresponding class c_a in the method signature for the argument, it is required that every successful ground substitution for V must be of class c_a, or a subclass of c_a.

- For each provider r where V appears in L, with a class conjunction c_r formed by the corresponding classes $c_{r_1},...,c_{r_n}$ in the method signatures for the arguments, it is guaranteed that every successful ground substitution in the feature for V is of every class c_{r_i}, or a subclass of c_{r_i}. Accordingly, it is of the class conjunction c_r.

- The class constraint $V : c_V$ in C, where $c_V = \{c_{V1},...,c_{Vn}\}$ represents a conjunction of classes associated with V, guarantees that every ground substitution for V for which the clause is successful will be of each class c_{Vi} or a subclass of c_{Vi} (equivalently, each ground substitution of V is of each $c_{Vi} \in c_V$).

It can be seen that there is a type-safety relationship between the classes of arguments where V appears, depending on whether it is an output or input, as well as the constraints on V. More precisely, the combined class of the variable provider should be such that all classes required by the variable consumers are guaranteed by the inherent classing resulting from the variable's use as a provider. A given assignment of a variable V to arguments then results in a valid set of classes which the variable V may adopt given the variable sharing. We

may reformulate definition 3.14 to take account of the classes of arguments among which a variable is shared, and the result of the sharing on the set of classes which the variable V may adopt. In doing so, we consider interactions between class constraints introduced in C and by the typed arguments in l_0 and L_B.

Definition 3.15. Consider a variable V appearing in a class-constrained feature $F = l_0 \leftarrow l_1, ..., l_n, c_1, ..., c_m$ as in definition 3.14. V is type-correct if:

- In L, the conjunction class c_r of formed from all arguments $c_r = \{c_{r1}, ..., c_{rn}\}$ where V appears as a provider must be a subclass of the class of each argument $c_{a1}, ..., c_{am}$ where V appears as a consumer, or
- In C, the class constraint $V : c_V$ restricts the set of possible ground substitutions from the providers from c_r to c_V, or one of its subclasses, *i.e.* where c_V is a subclass of c_r. Then, c_V (each c_{Vi}) must be a subclass of the class of each argument $c_{a1}, ..., c_{am}$ where V appears as a consumer.

Note that the class restriction $V : c_V$ here acts as a restriction on the set of possible output substitutions for V after a method invocation using V as an output, enabling the restricted id-terms to be used in a type-safe manner for the input arguments of a later method expression[2]. We can therefore compare the class restriction with the output classes c_{ri}, since they both impose the same restriction on the class of the variable V. The difference between the classes guaranteed by an output method and required by an input method can be seen as a sublattice of the lattice formed by the set of classes and the subclass relationship between them. We denote this lattice of valid classes which V may take as $G(V)$.

Definition 3.16 (Lattice of permissible classes for a variable in a class-constrained feature). A variable V appearing in a class-constrained feature $F = l_1, ..., l_n, c_1, ..., c_m$ may take any of the classes in the set $G(V)$, where $c \in G(V)$ if: (i) c is a superclass or equal to of the combined provider class c_r, or a superclass of the class constraint c_V; (ii) c is a subclass or equal to each consumer class c_{ai}.

$G(V)$ then defines the type-safety of the variable assignment of V to arguments of methods in a clause. As the clause is refined during induction, the lattice is further constrained and shrinks. Where $|G(V)| = 0$, there are not longer any valid ground substitutions for V. Where $|G(V)| > 0$, V is type-correct. By checking $G(V)$ for each variable V in a CORLOG feature, we may verify its type-correctness. We will take advantage of these type criteria during the formulation of the induction method in chapter 4. Variable unification, instantiation and the refinement of class restrictions all use this rule to test type-validity.

3.4 Domain descriptions: The role of metaknowledge

It is well accepted that when data mining in relational databases, the careful construction of background knowledge is crucial to the success of a multi-relational data mining task, but is a little-understood step in the process of applying ILP [131]. Successful data mining in relational databases usually involves the user supplying the learner with some higher-level description of the data which the miner can exploit to benefit the learning process. We call this model the *domain model*, or *domain description*, since it is intended to express important aspects of the domain being modelled, in particular its data relationships. As in data modelling, the domain model is an key engineering artefact in the process of data mining. In ILP, it provides a separation of the data

[2]Where the feature is evaluated left-to-right in the interpreter, this may require the class restriction to appear after the variable is first introduced.

level from the mining level, aiming to describe the data independent of the operation of the mining system. The form of such descriptions vary from learner to learner, but they all serve the purpose of informing the behaviour of the system for a specific task and in a given domain by specifying the data relationships in it. The term *domain* was defined in section 1.2 as the context from which data is taken for a data mining problem. Similarly, *domain analysis* refers to the identification of the information which is perceived to be important about a domain. In this thesis, the domain model is expressed in an object-based description language for *metaknowledge*. Metaknowledge exists in traditional ILP systems via forms of declarative bias introduced earlier, such as mode and type declarations.

Our framework so far has considered the representation of database facts and queries and object molecules, and how terms in these are constrained by a class, extended to a parametric framework. Central to this framework is the notion of a method signature. Method signatures define the classes taken by method arguments, and thus form an important and fundamental form of metaknowledge. Like other forms of metaknowledge, they incorporate additional information about the domain which serves to constrain the hypothesis language. They introduce the notion of type-safety in a feature. A feature is said to be type-valid if each literal in the relational part, together with the class constraints in the constraint part, conforms to its corresponding method signature. More generally, a feature can be determined to conform to a metaknowledge declaration, *i.e.* metaknowledge is framed as a set of domain-specific conditions which each feature must respect, reducing the size of the hypothesis space and defining new forms of redundancy and inconsistency in a feature given assumptions about its data relationships. If the conditions represented by the metaknowledge are satisfied by a feature, the feature is said to *respect* the metaknowledge. Otherwise, the feature *violates* it. In F-Logic, metaknowledge validity is enforced by the model theory of the language in the form of satisfaction rules, described in section 2.4.5. The proof theory then incorporates constraints on the derivation rules based on the metaknowledge.

The approach presented in this thesis extends this meta-knowledge typically found in an object database to a superset suitable for guiding an inductive learner. Meta-knowledge is therefore seen as a means of constraining the hypothesis space, the inductive bias being restricted to only metaknowledge-repecting features. Two kinds of metaknowledge are considered. *Method metaknowledge* concerns properties of a method and relationships between the host, input and output arguments. *Class metaknowledge* concerns relationships between classes and imposes additional semantics on class structure beyond subclassing relationships.

3.4.1 Method metaknowledge

Method metaknowledge has a number of forms in CORLOG. We can either consider the method as a whole, as with signature molecules, or consider the relationships between the arguments. In this thesis, when considering relationships between arguments we consider only the relationship between the host argument and the output. We could extend this framework to consider relationships among arbitrary sets of arguments, assuming constant values for the remaining arguments to form a projected version of an original method. Since the individual-centredness of the object model typically results in methods with few arguments, we restrict our attention to the cases above, though the inductive process is easily adapted to these projected forms. In induction, method metaknowledge introduces specific rules concerning the form of new literals (molecules) which are added to a constructed query. A summary of method metaknowledge is presented in table 3.1.

We have already considered metaknowledge in the form of signature molecules. Since method signatures are an inherent part of F-Logic, the model theory and proof theory describing their interaction with deduction

Declaration	*Meaning*
	Method declarations
$\mathsf{method}(c_0, m, a, t, k)$	Specifies a method index k, for method m, operating on objects of class c_0, with argument specifications $a = [\mathsf{cm}(c_1, m_1), ..., \mathsf{cm}(c_n, m_n)]$ taking classes c_i and modes m_i. Additional declaration tags appear in the list t.
	Multiplicity
	For a set of literals matching the enclosing method,
$\mathsf{fromcard}(r_1, r_2)$*	the number of different substitutions for the inputs is in the range and $n(r_1, r_2)$
$\mathsf{tocard}(r_3, r_4)$*	the number of different substitutions for the result is in the range (r_3, r_4). Where a maximum is ∞, it is represented in the syntax by the symbol inf.
	Orders
$\mathsf{wellorder}(L)$*	*Well order.* Defines that, for a method M, and $L = [v_1, ..., v_n]$, the resulting well order $<_P$ is such that, for each v_i, $v_i <_P v_j$.
$\mathsf{partialorder}(L)$*	*Partial order.* Defines that, for a method M, and $L = [v_{g_1} : v_{s_1}, ..., v_{g_n} : v_{s_n}]$ specifies that for each i, $v_{g_i} <_M v_{s_i}$.

Metaknowledge appearing as tags in the method declaration are marked with an asterisk (*).

Table 3.1: Method metaknowledge in CORLOG

is well-defined and has been reported in section 2.4.5. We begin by establishing some definitions regarding coverage by signature expressions.

Definition 3.17 (signature-respecting feature). A feature $C = M_0 \leftarrow M_1, ..., M_n$ is *signature-respecting* if, for each M_i, $0 \leq i \leq n$ of the form $O_i[M_i(I_{i_1}, ..., I_{i_m}) \rightarrow R_i]$, there exists a matching method signature MS_i of the form $Q_i[M_i(C_{i_1}, ..., C_{i_m}) \rightarrow S_i]$, such that each O_i is constrained to be an instance of the class Q_i, each I_{i_j} is constrained to be an instance of the class C_{i_j} and each R_i is constrained to be an instance of the class S_i. We say that the method M_i *conforms* to the signature MS_i.

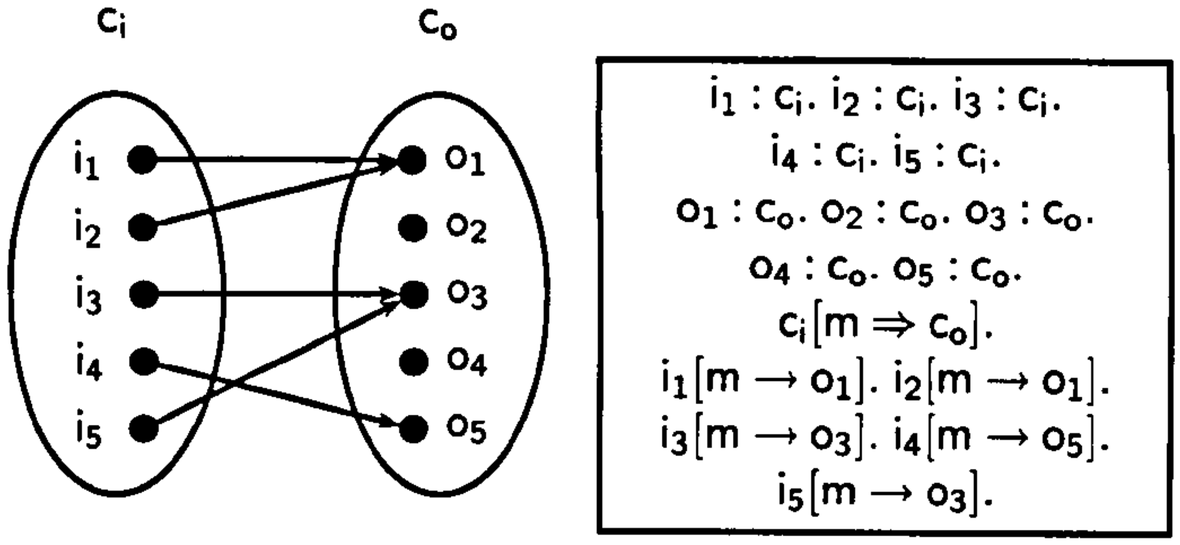
In the remainder of this section, we consider multiplicity and functionality constraints on the success set of a method call given the possible successful substitutions of the arguments. We then continue the discussion of constraints by considering the role of metaknowledge in defining orderings over data expressions. Multiplicity and order-yielding properties are defined as part of method declarations, which in practice serve to type methods in the induction process. A method declaration is simply a ground fact in CORLOG defining the relevant data structures.

Definition 3.18 (CORLOG method declaration). A CORLOG method declaration is a fact of the following form:

$$M = \mathsf{method}(c_0, m, [\mathsf{cm}(c_1, m_1), ..., \mathsf{cm}(c_n, m_n)], [t_1, ..., t_l], k)$$

where:

- The first three arguments uniquely identify a method signature in the database. m is a method operating on a host object of class c_0. Each argument $\mathsf{cm}(c_i, m_i)$ is associated with a class c_i from the database and m_i, equal to the symbol in or out, defines the mode.
- k is a *unique* number associated with the method, inducing an order over the set of all methods.



A bipartite graph representing the mapping introduced by the method m between objects of the host object class c_i and objects of the output class c_o. The relevant fragment of the program is shown to the right.

Figure 3.2: Multiplicity of a data relationship, described using a bipartite graph

- $t = [t_1, \ldots, t_l]$ is a list of atoms. Each element t_i acts as a tag. The presence of a tag in t indicates some metaknowledge property of the method.

3.4.2 Multiplicity and functionality

A commonly-used data relationship, both in database schemas and in ILP, is the notion of multiplicity on a composition, or 'part-of' relationship. A *multiplicity relationship* comprises two *cardinality bounds*, defining a range in the positive integers. Booch [11] describes cardinality as the number of links between each instance of the source class and instances of the target class. There is a bound on the source class (the *from-cardinality*) and on the target class (the *to-cardinality*).

In the CORLOG framework, a composition takes the form of a method. A method maps a single host object to a set of resulting successful output objects. This mapping can be visualised as a bipartite graph as in figure 3.2, mapping objects belonging to a source class c_i to objects belonging to a target class c_o. It can be seen from the graph that each i_j maps to at most one o_k, although there exist output values in o_k which are mapped from more than one i_j. The cardinality of the association as it appears in the database is then described by the range of out-degrees of nodes in c_i and the in-degrees of nodes in c_o. Metaknowledge specifies appropriate bounds on these values arising from the semantics of the data in terms of a multiplicity bound describing substitutions for its input arguments and result substitutions r.

Definition 3.19 (multiplicity, cardinality constraint). For a given database D and a method M, we define the value $OB_D(i, M)$ as the number of successful distinct output substitutions for a *tuple* of input arguments i under the method M in D. Similarly, $IB_D(o, P)$ is the number of distinct input substitutions resulting in the output argument o under M in D. A pair of multiplicity constraints

$$\mathsf{fromcard}(r_{\mathsf{from,min}}, r_{\mathsf{from,max}}), \mathsf{tocard}(r_{\mathsf{to,min}}, r_{\mathsf{to,max}}) \tag{3.8}$$

specified in the metaknowledge for a method M guarantees:

$$r_{from,min} \leq \min_i OB_D(i,M) \tag{3.9}$$

$$r_{from,max} \geq \max_i OB_D(i,M) \tag{3.10}$$

$$r_{to,min} \leq \min_o IB_D(o,M) \tag{3.11}$$

$$r_{to,max} \geq \max_o IB_D(o,M) \tag{3.12}$$

The maxima $r_{from,max}$ or $r_{to,max}$ may be set to the symbol inf, in which case no maximum is set, or equivalently, the maximum ∞ is adopted.

Example 3.20. The strictest bounds possible for the mapping in figure 3.2 are fromcard$(1,1)$ and tocard$(0,2)$.

In object modelling, the cardinality bound of each side of an association is characterised by the notation $a..b$ for a from-bound a and a to-bound b, in which numbers (often 0 or 1) appear for a and b specifying a single value for the bound. n or $*$ denotes the absence of a bound. The constraint adopted extends these bounds by modelling two 1-to-(n,n') constraints simultaneously, for a range of values n to n' inclusive. We choose instead to denote the bound on each side of the multiplicity relationship by $r_{from,min}..r_{from,max}$ for the from-cardinality bound and similarly for the to-cardinality bound.

Participation restrictions are a special case of multiplicity restrictions which are commonly defined in database models based on the entity-relationship model [15]. A participation declaration specifies whether an association, here situated as a method call, is defined for each instance of a class. Where each instance of a class results in a successful method call, it is termed *total*. If a method call is not total, it is *partial*. In the framework of cardinality declarations, a partial method call is represented with a minimum to-cardinality of 0. If it is total, the minimum to-cardinality is 1 (or more). Furthermore, multiplicity may be extended to model functionality of an association or method. An association (*i.e.* method) may be declared as having the property *functional* or *inverse-functional*. In both cases, an attribute of each object of a given class acts the equivalent of a primary key, providing a unique identifier for the object in the system (other than its object indentifier).

Functionality may be easily characterised in terms of the multiplicity relationships as described. A method M is functional if for every substitution of the input arguments in M, such that the method succeeds, there is a *unique* value returned from the method. This corresponds to specifying a to-cardinality bound of 0..1. Equivalently, each possible output value may be mapped from at most one tuple of input values. A to-cardinality bound of 1 would further require that every possible output value is mapped. Conversely, a method M is inverse-functional if for every possible output value returned from the projection, there is a *unique* substitution of the input arguments in M. Similarly, each possible tuple of output values maps to at most one output value, and so a from-cardinality bound of 0..1 represents this relationship. A from-cardinality bound of 1 requires that every tuple of inputs produces a mapping. The notions of injection, surjection and bijection can also be conveniently modelled, using to-cardinality bounds of 0..1, 1..$*$ and 1 respectively.

Multiplicity relates to the concepts of determinacy and recall in ILP systems. For example, a method defining a to-cardinality of $m..n$ may only be repeated at most n times in an induced feature for the same input bindings. Similarly, a recall number of n in traditional ILP systems bounds the determinacy — the number of successful substitutions given a set of input variables — of a predicate, and ensures they are not repeated more than n times per set of input substitutions. Other forms of bias related to multiplicity exist in particular ILP systems. We have discussed before the restriction of ILP searches to clauses which are determinate. A

clause $h \leftarrow b_1,...,b_n$ is determinate if and only if for all substitutions θ that ground h, there exists at most one substitution σ such that $b_1\theta\sigma,...,b_n\theta\sigma$ succeeds in the background knowledge B [29]. In the framework above, this assumes that each method has a to-cardinality bound of 0..1. Likewise, a predicate is said to be functional if, for some rearrangement of its arguments, $X = Y \leftarrow p(X_1,...,X_n,X), p(X_1,...,X_n,Y)$ [29]. Taking this further, Flach [44] described INDEX, a system for inducing a set of generalised constraints of this form. In it, a *functional dependency* is a clause of the form $\overline{Y} = \overline{Z} \leftarrow r(\overline{X},\overline{Y},\overline{W_1}), r(\overline{X},\overline{Z},\overline{W_2})$ where $\overline{V}$ denotes a vector of variables. The arguments taken by $\overline{Y}$ and $\overline{Z}$ are then functionally dependent on the arguments taken by $\overline{X}$, corresponding to a method in the above formulation. $\overline{W_1}$ and $\overline{W_2}$ then are analogous to the ground arguments in the projection of a method.

The legality of a feature with respect to its multiplicity metaknowledge is defined in terms of subsets of those literals which share the same method symbol.

Example 3.21. Consider the following method from a telephone book database.

$$M = \mathsf{A}[\mathsf{lookup} \rightarrow \mathsf{N}], \mathsf{A} : \mathsf{name}, \mathsf{N} : \mathsf{number} \tag{3.13}$$

Assume the following declarations have been assigned to M: $\mathsf{fromcard}(0,*)$ (a person may have any number of telephone numbers, including none) and $\mathsf{tocard}(1,1)$ (each telephone number corresponds to exactly one person) *i.e.* the from-cardinality bound is 0..$*$, the to-cardinality bound is 1, and the method is necessarily functional. For two literals M and M' appearing in a feature, if

$$M = \mathsf{alice}[\mathsf{lookup} \rightarrow 6841] \tag{3.14}$$
$$M' = \mathsf{bob}[\mathsf{lookup} \rightarrow 6841] \tag{3.15}$$

then the inclusion of M' breaks the from-cardinality bound. However, if

$$M = \mathsf{X}[\mathsf{lookup} \rightarrow 6841] \tag{3.16}$$
$$M' = \mathsf{bob}[\mathsf{lookup} \rightarrow 6841] \tag{3.17}$$

then the inclusion of M' does not break the from-cardinality bound, since X could legally be substituted for bob.

It could be argued that in the second pair of literals, where X is substituted for bob, M becomes redundant. However, the from-cardinality only ensures that the literal M is equivalent to testing that the variable X is bound to the value bob. Accordingly, M and M' are not mutually redundant. Similar results follow for to-cardinality. Having defined the multiplicity framework, and therefore the functionality framework, we are in a position to define a clause in terms of whether it respects a given multiplicity declaration

Definition 3.22 (multiplicity-respecting feature). Consider a feature $F = \mathsf{M_0} \leftarrow \mathsf{M_1},...,\mathsf{M_n}$, which has no duplicate literals and uses a set of corresponding method symbols $m_1,...,m_n$. For a method symbol m_i, let L_i denote the set of literals using the method symbol m_i. Let $frommax_i$ and $tomax_i$ denote the *maximum* from- and to-cardinality bounds, respectively, of method m_i. F is then *multiplicity-respecting* if for each m_i:

- *(from-cardinality).* For each m_i, among the subset L' of literals in L_i involving method m_i, there are at most $tomax_i$ literals in L' with the same substitutions for input arguments.

- *(to-cardinality).* For each m_i, among the subset L' of literals in L_i involving method m_i, there are at most $frommax_i$ literals in L' with the same substitutions for output arguments.

Observe that the definitions take no account of the minimum cardinality specification. This is because the presence of a minimum cardinality does not *require* a feature to include it. That is, the inclusion of an association is optional in a feature. The minima are not used in the inductive system proposed in chapter 6 and therefore may be omitted so that fromcard and tocard take one argument only. Secondly, by assuming that there are no repeated literals in the feature, literals with the same input arguments necessarily map to different outputs. We can therefore assume that more than n repeated literals with the same substitutions for input arguments map to n distinct output values, and where $n > m$ for some maximum cardinality bound m, the feature violates the multiplicity metaknowledge. Conversely, literals with the same output arguments necessarily map to different input arguments.

3.4.3 Orderings over arguments

Orderings are an additional form of metaknowledge which may be imposed on classes of id-terms in the database. For example, the set of integers under the less-than relation may be considered to be ordered. In the terminology of an object-oriented deductive database, a class of objects C (*e.g.* integer) is ordered by a method M (*e.g.* that defined by the method signature integer[lessthan(integer) ⇒ bool]. Orderings offer two benefits for induction; firstly, they allow a more domain-specific form of refinement by replacing arguments in method calls to give rise to a more specific method expression, and secondly, they allow the detection of redundancy among method expressions in the same feature.

Informally, orderings introduce the idea of some sequence or arrangement of the elements of a set. We identify two types of orders over a set, in terms of an abstract relation $\sim$. A *partial order* does not necessarily define whether $x \sim y$ for every pair of objects in the set. *Well-orders* form a linear order, *i.e.* a set in which the each element can be considered to have a successor element (though there need not be a predecessor for each element). These orders are specified as metaknowledge according to the syntax shown in table 3.1.

The relation $\sim$ can be considered to exist between two arguments in a method. A method which gives rise to at least a partial ordering is termed an *order-yielding method*. Such methods are useful because they can be linked to orderings over method calls which exhibit subsumption properties. A simple example might is $M =$ X[lessthan(Y) → true], which succeeds if $X < Y$. Consider a query which, when executed for each individual in the database, returns a set of bindings for some variable X of type integer. By adding a method expression which grounds Y to some constant term y in M, we test the values of each X-binding. A method expression $M_1 =$ X[lessthan(y) → true] then succeeds for fewer examples than an expression $M_2 =$ X[lessthan(y′) → true] where y < y′ for ground y and y′. M_2 thus subsumes M_1, and M_1 is a stricter expression than M_2. In general, a method expression M_1 is stricter under an order-yielding method M than another method expression M_2, where M_1 and M_2 share a host object O of class C, if M_1 is guaranteed to succeed for a subset of fewer host objects O than M. An strictness or subsumption ordering is thus defined over methods in M, following directly from the transitivity property of orders. We consider two forms of methods. M_1 takes two inputs, whereas M_2 takes a single input and produces an output.

Proposition 3.23 (subsumption of order-yielding methods). *For any method* $M_1 =$ X[m(Y) → z] *(host/input) or* $M_2 =$ X[m → Y] *(host/output) which is order-yielding and such that* X *and* Y *are of class* C*, and some ground term* z*,*

Declaration	*Meaning*
	Class hierarchy
sc :: c	subclass expression
	Abstract class
abstract(c)	states that a c may not be instantiated
	Dimensional inheritance
disjoint($[c_1,...,c_n]$)	Denotes that no object may belong to any pair of classes in c_i.

Table 3.2: Class metaknowledge in CORLOG

- *For methods of the M_1 pattern, (i)* $X[m(y) \rightarrow z]$ *is subsumed by (is stricter than)* $X[m(y') \rightarrow z]$, *wherever* $y[m(y') \rightarrow z]$; *(ii)* $y'[m(X) \rightarrow z]$ *is subsumed by (is stricter than)* $y[m(X) \rightarrow z]$, *wherever* $y[m(y') \rightarrow z]$.
- *For methods of the M_2 pattern, (i)* $X[m \rightarrow y]$ *is subsumed by (is stricter than)* $X[m \rightarrow y']$, *wherever* $y[m \rightarrow y']$; *(ii)* $y'[m \rightarrow X]$ *is subsumed by (is stricter than)* $y[m \rightarrow X]$, *wherever* $y[m \rightarrow y']$.

Features are refined (made more specific) by replacing a method expression, with a stricter one based on an associated method M. Since literals represent conditions on the bindings of variables, this reduces the set of objects for which the method succeeds[3]. Such methods are known as refinable methods, which are formalised as follows:

Definition 3.24 (refinable method, least refinement). A data expression $M_1 = O[M(I_1,...,I_n) \rightarrow R]$ is *refinable* if there exists a order-yielding method signature M such that M_1 matches M. A *refinement* of M_1 produces a data expression M_2 matching M but which is subsumed by M_1, according to proposition 3.23. A *least refinement* of M_1 produces a data expression M_2 matching M but for which there is no data expression M_3 matching M such that M_1 subsumes M_3 and M_3 subsumes M_2.

The requirement to match with M defines a subsumption hierarchy over the set of possible arguments of the method M_1, equivalently defining an ordering $\leq_M$ over the substitutions for variable arguments in M such that, for example, for any two method calls with identical host object and output substitutions, M_i with argument substitution x_i and M_j with argument substitution x_j, the set of solutions of O for M_i is guaranteed to be a subset of solutions for M_j with respect to the implied argument ordering $x_i \leq_M x_j$. The assertion wellorder($[v_1,...,v_n]$) appearing in a method declaration states that where $\leq_M$ is a wellorder underlying an order-yielding method in M, *i.e.* for each v_i appearing before each v_j in the list, $v_i \leq_M v_j$. The assertion partialorder($[v_{g_1} : v_{s_1},...,v_{g_n} : v_{s_n}]$) states that where $\leq_M$ is a partial order underlying an order-yielding method in P, for each i, $v_{g_i} \leq_M v_{s_i}$.

3.4.4 Class metaknowledge

There are two forms of class metaknowledge exploited in our object model. We consider firstly metaknowledge which concerns individual classes, and secondly metaknowledge which elaborates on the inheritance relationship between two classes. For the former, we consider abstract classes, and for the latter, the notion of dimensional inheritance. We consider each in turn. Table 3.2 summarises this metaknowledge.

An abstract class is a class which is designed to be treated as a superclass; no object in the system is a direct member of an abstract class, but instead classes inherit from abstract classes. They are used to represent

[3]Note that such refinements require substitution of ground id-terms for other ground id-terms and is therefore not considered a substitution as defined previously.

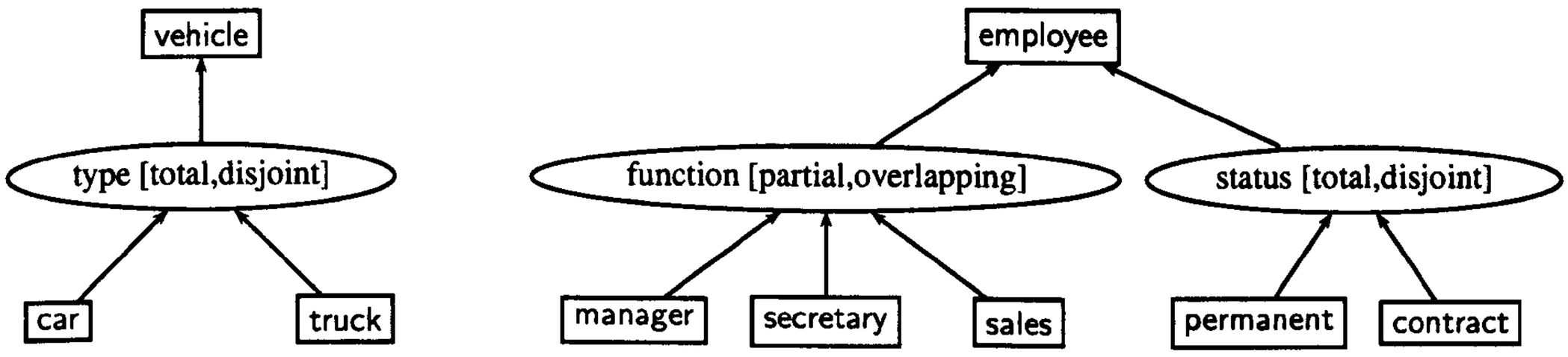


Figure 3.3: An example of dimensional inheritance

interface or partial implementation which is shared by a number of other classes via inheritance. Alternatively, abstract classes may be used as mixin classes, defining extra functionality to existing classes. The opposite of an abstract class is a *concrete class*, *i.e.* a class for which objects may be instantiated.

A class C is asserted to be abstract with the fact abstract(C). The notion of abstract classes are of most use in refining clauses, although a clause, and indeed a program or database, may be said to be *abstraction-respecting* if it contains no ground instances of a class which is abstract.

Disjoint and dimensional inheritance places restrictions on the set of inheritance relationships from a specific class C in the database to a set of its subclasses C_i'. A family of these relationships can be defined in terms of the *disjointness* and *totality* of the inheritance.

Definition 3.25 (disjointness, totality). Where the relationship between C and C_i' is *total*, each object of class C must also be an object of *at least* one of the subclasses C_i'. If a relationship is not total, it is *partial*. Where the relationship between C and C_i' is *disjoint*, an object must be of *at most* one of the subclasses C_i'. If this is not true, the relationship is *overlapping*.

Disjoint and overlapping inheritance is a common database modelling feature which grew out of a desire to integrate inheritance semantics into the original entity-relationship model [15] of database models, one of the bases for object-oriented database models. The extended entity-relationship model [136] included provision for total disjoint and overlapping generalisation.

Observe that it is sufficient to cover these cases by declaring classes as abstract and/or disjoint. We may reformulate the notion of disjointness by defining a set of classes $C = \{c_1, ..., c_n\}$ to be disjoint with the meta-knowledge declaration disjoint$[c_1, ..., c_n]$ This means given any pair of classes (c_i, c_j), $c_i \in C$, $c_j \in C$, $c_i \neq c_j$, no object may be an instance of both c_i and c_j, including its subclasses. For a set of subclasses with a common superclass, totality is assured by setting the superclass to be abstract. Where it is not, the relationship is partial. Disjointness may be specified by setting the set of subclasses to be disjoint. Where this is not specified, the relationship is overlapping. Note that this scheme for specifying dimensional inheritance is more flexible, since subsets of the subclasses may be specified disjoint.

Figure 3.3 illustrates two class structures which are constrained by dimensional inheritance. Classes are denoted by rectangular boxes and discriminators by ovals. Inheritance relationships are denoted by arrows. The superclass vehicle has two subclasses car and truck. The inheritance is mediated by a discriminator type. *type* is total and disjoint, and so each instance of vehicle must therefore be an instance of car *or* truck. The vehicle class then abstracts elements of the implementation common to car and truck, such as the number

plate. car and truck then add elements specific to cars, such as the number of passengers, and truck, such as the number of axles or tonnage. The second example shows two discriminators working together. In it, an employee must be either permanent or contract according to the discriminator *status*, but an employee may be any of manager, secretary or sales, or indeed, none of them. The superclass vehicle is then abstract, while the partial constraint on the discriminator *function* means that employee is not abstract.

Returning to the reformulation of this approach to the use of metaknowledge declarations using abstract and disjoint classes, we define what it means for a feature to respect class metaknowledge.

Definition 3.26 (abstract- and dimensional-inheritance-respecting feature). A feature $F = \mathsf{M}_0 \leftarrow \mathsf{M}_1, \ldots, \mathsf{M}_n$, is *abstract-* and *dimensional-inheritance-respecting* with respect to abstract class declarations of the form $\mathsf{abstract}(\mathsf{a}_1)$ to $\mathsf{abstract}(\mathsf{a}_m)$ and disjoint class declarations of the form $\mathsf{disjoint}(\mathsf{d}_1)$ to $\mathsf{disjoint}(\mathsf{d}_k)$ if for each variable V in F belonging to a non-empty set of classes C_V, it holds that (i) where a class in the conjunction C_V is a member of some d_i, no other class in C_V is a member of d_i also. (ii) no class in the conjunction C_V is declared abstract with a declaration $\mathsf{abstract}(\mathsf{C}_V)$.

3.5 Conclusion

This chapter has presented CORLOG, a language suitable for mining deductive databases expressed in the object model, in which each individual is represented by an object taking a class, and each class has methods defined on its data. The clausal language has three main aspects. The first is the object part, in which new variables are introduced by calling typed methods on variables already in the clause according to type safety constraints. The second is a constraint part, which tags terms with classes, which restrict the values which variables in the clause may take. The third part is meta-knowledge about the classes and objects, which define the validity of a clause and inform feature construction during learning in using extra declarations in terms of the object model.

We therefore view object deduction and induction as being a constrained version of their first-order counterparts, the first form of constraint being the class constraint and the second the validity imposed by meta-knowledge in the database. Adopting such an approach introduces a strong bias. In particular, the requirement for each method call to take a host object presents a form of the individual-centred representation, providing a practical trade-off between expressivity and the size of the search through the hypothesis space, while still allowing reasoning and induction in data which naturally forms inheritance hierarchies.

In particular, the adoption of the object model provides a number of interacting modelling constructs, which are typically characterised as constraints or restrictions on a valid feature. Principally among these are the introduction of a class hierarchy and object membership to classes, in which taxonomies of objects can be expressed; method declarations which define for which argument a method expression can be determined to be type-validity; the abstraction of structure in parameter classes; and a family of metaknowledge restrictions on methods, defining additional forms of redundancy and subsumption which can be exploited by the learner. Together, these constructs closely define the data model and the behaviour of the method expressions which are valid in the hypothesis language. Through this, CORLOG introduces close restrictions on the hypothesis space.

In the next chapter we build on the language bias introduced by the object model, to consider approaches to induction in CORLOG and their implementation in an ILP system by introducing structure and a search method over the space of CORLOG features, leading to an inductive process informed by the metaknowledge declarations defined in this chapter.

Chapter 4

Induction in object logic

The previous chapter considered reasoning in a logic constrained by the object model, describing a method of *deduction*. In this chapter we build on these definitions in order to develop techniques that support *induction* in the object model, and in particular propositionalisation, the approach to inductive logic programming considered in this thesis. In section 4.1, we firstly review general approaches to the task of inductive logic programming in section, specific settings for ILP and the ways to impose a generality structure on the hypothesis space. In section 4.2, we then consider means of traversing this structure, with a particular focus on the process of refinement. We then move to the specific case of induction in CORLOG in section 4.3, considering forms of generality and refinement techniques specific to the object model introduced. In this way, we present a means of performing ILP for object-oriented data mining. Section 4.4 concludes.

4.1 Inductive logic programming

Induction is the learning of a general theory from specific examples, aiming to find general properties from specific instances of those (as yet unknown) properties. It is therefore the dual of deduction, which generates specific examples given a general theory. Induction has been a topic of inquiry for centuries and is often seen as a main source of scientific knowledge [111].

Example 4.1 (Induction of rules concerning measles [111]). A good example of induction is given in [111] in which a hospital's records provide examples from which we want to find general rules concerning which symptoms indicate which diseases in patients. These rules tell us general patterns about everyone in the hospital's records. Consider the case where it is known that every patient in the hospital has fever and red spots suffers from measles, the general rule 'if someone has a fever and red spots, he or she has measles' might be inferred. Similarly, if every patient diagnosed with measles also has red spots, we can infer the rule 'if someone has measles, he or she will have red spots'. Such rule learning can be supplemented by *background knowledge* relevant to the learning task. Each patient suffering from measles may be infected by a virus a or b, and no patient with virus a or b has measles. Furthermore, both virus a and b belong to a virus family c. With the aid of this background knowledge, the two simple rules consisting of 'if someone is infected by virus a, he or she has measles' and the same for virus b can be replaced with the stronger rule 'if someone is infected by a virus x from a family c, he or she has measles', which has more predictive power.

In inductive learning, we are given a set of observations, and, together with a body of background knowledge, we wish to find a theory which generalises those instances. The role of the background knowledge in the inductive learning process is pivotal. The study of inductive learning with respect to background knowledge came about from the realisation that the goal of artificially intelligent systems — to learn from experience and learn and adapt from situations it encounters — required a degree of inductive power. The inclusion of ground examples without background knowledge limited this power in terms of the strength of knowledge which could be inferred. The natural incorporation of background knowledge in inductive systems incorporating logic saw a great degree of interest in logical induction in the field of artificial intelligence. As a result, logical techniques to inductive learning grew, alongside a wide variety of non-logical machine approaches to induction, such as neural networks and genetic algorithms. The formalisation of induction for clausal logic was introduced as early as 1970, when the first work on the normal ILP setting was done by Plotkin [113, 114, 115], in which examples as ground clauses were generalised by computing their least general generalisation. Similar work on least general generalisations was undertaken by Reynolds [118] at the same time. This early work provided the framework of θ-subsumption for decidable testing of entailment and structuring between hypotheses, leading to the adoption of clauses for expressing examples, background knowledge and induced theory. Accordingly, many ILP systems have adopted the general-purpose logic programming language Prolog as a computational and representational basis.

Concept learning is an example of a logical approach to induction [96]. In it, a general definition of some concept is induced from a set of examples labelled as belonging to or not belonging to the concept. By searching through a space of hypotheses defined by some representation language and language bias, a hypothesis is chosen which best fits the examples given. The representation language and language bias therefore define the set of hypotheses — the *hypothesis space* — which can be considered by the ILP system and therefore can be learned.

Muggleton [108] defined the field of inductive logic programming (ILP) as the intersection of machine learning — the branch of artificial intelligence which studies learning — and logic programming. Inductive logic programming (ILP) has become considered as the task of concept learning where the representation language is Horn clausal logic, the basis for logic programming languages such as Prolog. Nienhuys-Cheng and deWolf [111] argued that the adoption of the clausal logic framework as a representation language for observations, background knowledge and induced theories has a number of important benefits. The basis of logic means that a well-developed body of concepts, techniques and results from mathematics can be applied to inductive logic programming. Logic also provides a unified and expressive means of representing the three ingredients of examples, background knowledge and the induced hypotheses. Finally, results expressed in logic are more easily interpreted by humans.

Since hypothesis spaces in logic programming representations can potentially be very large, it is impractical to test each hypothesis individually. ILP systems therefore perform a search over a hypothesis space, necessarily defining a generality ordering over the hypothesis space and a means of searching through the space using this ordering. Many inductive logic programming approaches and techniques exist today, for different choices of the representation language, bias, the ordering of the hypothesis space and the method of searching it, and these choices have relied on different formulations of induction.

4.1.1 Common settings

We study the process of machine induction in terms of its learning task, starting from a general formulation and refining it to a more specific, inductive setting. We study machine induction, and particularly inductive logic programming, as a specific example of a data mining technique. Accordingly, we consider the following definition of a data mining task, due to Mannila [91]:

Definition 4.2 (the task of data mining [91]). Given a hypothesis language $\mathcal{L}$, a data set D, and an interestingness criterion Q, the *task of data mining* is to find the set of hypotheses $T(\mathcal{L}, D, Q) = \{h \in \mathcal{L} | Q(h, D) \text{ is true}\}$.

T therefore represents the set of hypotheses in the language which are considered interesting. We may refine the task to find all these hypothesis, or a subset of T, or simply one of them. The choice of the Q criterion is left deliberately open and may be defined as a wide variety of possible criteria, for example whether a threshold on accuracy is reached by the hypothesis or the number of examples covered. In adopting an ILP approach, we place restrictions on the form of $\mathcal{L}$ and D, namely that they should be expressed in a first order logic, such as the typical first-order Horn clausal logic employed in ILP or the CORLOG framework. The adoption of a language bias and metaknowledge further restricts $\mathcal{L}$.

This definition of data mining, however, misses a number of key aspects of the inductive logic programming task, omitting the role of both background knowledge and logical implication. We therefore adopt more specific definitions from the ILP literature to cover logical inductive learning. We arrange these tasks into a collection of common learning *settings*. Such settings naturally divide along a number of principal features, the two most recognised being the difference between predictive and descriptive induction, and the choice of example representation. We consider each before adopting one for the approach taken in this thesis. For more detail, the reader is referred to the general comparison given in the tutorial introduction [48] and also to an earlier comparison in [94] in which conceptual inductive learning is separated into concept acquisition tasks (learning from examples) and descriptive generalisations (learning from observations).

Predictive vs. descriptive ILP

Predictive induction is the induction of hypotheses which discriminate between individuals tagged with different class labels. As such, predictive induction is associated with supervised problems in machine learning such as the learning of classification rules [117, 105, 101] and decision trees [10] or regression [59]. On the other hand, *descriptive induction* aims at the detection of regularities in a set of examples which do not possess a class label. Descriptive induction is therefore most similar to tasks learning from positive examples only, such as clustering [68], the learning of clausal theories [31] and association rules [36, 47], and subgroup discovery [140]. Flach and Lavrač [48] formalise the learning tasks of predictive and descriptive induction. Firstly, predictive induction is a discriminative task, and as such, takes place with respect to a set of positive and negative examples, between which we wish to discriminate. We use the conditions of completeness and consistency.

Definition 4.3 (predictive induction [48]). Let P_F and N_F be sets of ground facts over a set of foreground predicates F, called the *positive examples* and the *negative examples*, respectively. Let T_B, the *background theory*, be a set of clauses over a set of background predicates B. Let L be a *language bias* specifying a *hypothesis language* $\mathcal{H}_L$ over $F \cup B$ (*i.e.*, a set of clauses. A *predictive ILP task* consists in finding a *hypothesis* $H \subseteq \mathcal{H}_L$ such that

- $\forall p \in P_F, T_B \cup H \models p$. Where this is true, we say H is *complete*.
- $\forall n \in N_F, T_B \cup H \not\models n$. Where this is true, we say H is *consistent*.

If H is both complete and consistent, we say H is *correct*.

For descriptive induction, classification is no longer the aim, and the notions of positive and negative examples no longer exist. Instead, we aim to find a hypothesis which is true for all examples.

Definition 4.4 (descriptive ILP [48]). Let E be a collection of evidence and let m_E be a model constructed from E. Le L be a language bias specifying a hypothesis language $\mathcal{H}_L$. A *descriptive ILP task* consists in finding a hypothesis $H \subseteq \mathcal{H}_L$ axiomatising m_E, *i.e.*, H is true in m_E (the *validity* condition) and $\forall g \in \mathcal{H}_L$: if g is true in m_E then $H \models g$ (the *completeness* condition).

Definitions 4.3 and 4.4 relate to definition 4.2; the hypothesis language $\mathcal{L}$ in definition 4.2 corresponds to $\mathcal{H}_L$ in definitions 4.3 and 4.4. Similarly, the data set D is represented by $P_F \cup N_F$ in the former and E in the latter. The quality criterion Q is correctness in the former and validity and completeness in the latter. A further common requirement on H appearing in alternative definitions [104, 87] is that the valid hypothesis H should be *maximally general*, *i.e.*, that no proper subset of H is valid and complete.

Learning from interpretations

Discussion so far has made the assumption of a uniform representation language for the examples, background knowledge and induced hypotheses. In some ILP settings such as that of de Raedt [29], the representation language for the examples ($\mathcal{L}_e$) may be different from the representation language for the induced hypotheses ($\mathcal{L}_h$). Examples are covered by a hypothesis under a coverage relation $c \subseteq \mathcal{L}_h \times \mathcal{L}_e$[1]. Decoupling these two representation languages permits the separation of ILP techniques into one of two prevalent approaches. In *learning from entailment*, $\mathcal{L}_h$ is the language of clausal theories (sets of sets of clauses) and $\mathcal{L}_e$ is the language of clauses. $(h, e) \in c$ if $h \models e$, *i.e.* the hypothesis h logically entails the example e. In *learning from interpretations* [33], $\mathcal{L}_h$ remains the language of clausal theories, but $\mathcal{L}_e$ is a language of Herbrand interpretations — the set of ground atoms constructed with the predicate, constant and function symbols in the alphabet, representing possible worlds by explicitly specifying all true facts in that world, meaning those which are not stated are taken to be false. Then, $(h, e) \in c$ if e is a model for h[2] The interpretation is then the set of all ground facts describing a particular example, and coverage may then be implemented by a simple subsumption test. A good further comparison of these and other settings is given in [31], which applies learning from interpretations to a descriptive ILP task.

Helft [55] strongly influenced the study of learning from interpretations with his *non-monotonic setting*. The assumption that anything not specified is false is known as the closed-world assumption which assumes that the given description of the world is not only true, but complete; it contains all information concerning the world [111]. On the other hand, in the non-monotonic setting it is assumed that all observations are completely specified. Furthermore, whereas in the usual setting an acceptable hypothesis H is such that it implies all the examples E (and possibly other examples), in the non-monotonic setting, if H were to imply examples not in

[1] De Raedt points out that in some sitations, the use of a relation may be too restrictive. Where probabilistic or regression models are used, using a real number to represent coverage would be more appropriate.

[2] A Herbrand interpretation I is a model for a clause c if and only if for all grounding substitutions θ of c, $body(c)\theta \subset I \rightarrow head(c)\theta \cap I \neq \emptyset$ [31]

E, H would no longer be valid, since these examples would falsify the hypothesis under the non-monotonic setting. H must therefore hold for *all* E, leading to a stronger validity requirement and more conservative properties. Non-monotonic semantics are among those De Raedt and Dehaspe argue are suitable for descriptive induction from interpretations [31]. A good comparison between the settings following from the normal and non-monotonic semantics is given in [104].

Briefly, we draw a distinction between batch and interactive learning in ILP. Batch learning assumes that all the examples are given at the outset, and may therefore apply statistical tests on the full example set and detect and reduce noise or other undesirable properties. Incremental learners, on the other hand, accept examples which are given one at a time. During learning, the learner adjusts its theory so that it it matches the examples given so far. Interactive systems [27], which are usually incremental, take this process one stage further, and can pose questions to the user in order to construct further examples during the search. Lavrač and Džeroksi introduce interactivity to previously-discussed settings by adding a current hypothesis $\mathcal{H}$, a (new) labelled example e, and an oracle (the user) willing to label examples as positive or negative and possibly other information such as the validity of generalisations which it produces. Clauses are added and deleted to form new hypotheses $\mathcal{H}'$ to make it correct, possibly shifting bias in the process. The early MIS system of Shapiro [128] employed the interactive setting, and the later CLINT [32] system uses integrity constraints in order to maintain a set of hypothesis languages, moving between them during search. In this thesis, we consider only batch learning.

The normal ILP setting

As noted by Lisi [87], the combination of these two commonly-adopted categorisations — descriptive vs. predictive and use of entailment vs. interpretations — for inductive logic methods leads to four logical frameworks for ILP. Of these, by far the most prevalent in the literature is predictive ILP learning from entailment, which has come to be known as the *normal problem setting for ILP*. It is this setting which is adopted for study in this thesis and adapted to the object model. The normal ILP setting is succinctly defined by Nienhuys-Cheng and de Wolf [111] as follows:

Definition 4.5 (the normal ILP setting [111]). *Given* a finite set of clauses $\mathcal{B}$ (background knowledge), and sets of clauses E^+ and E^- (positive and negative examples), *find* a theory Σ such that $\Sigma \cup \mathcal{B}$ is correct with respect to E^+ and E^-.

Note that this definition is restricted to problems of two classes, whereas the general predictive task concerns only the existence of different class labels. We may adapt the definition by, for example, considering one-against-all learning, in which a prediction problem with k classes is reduced to k two-class problems. For a different class c_k in each k, examples of class c_k are adopted as the positive examples and all remaining examples are adopted as the negative examples.

4.1.2 Structure: Ordering hypotheses by generality

A naive approach to induction would involve constructing all possible hypotheses in the language $\mathcal{L}$ and determining for each whether the quality criterion Q applies. For the large, expressive hypothesis spaces used in ILP, the cost of checking Q for each hypothesis is often too great. Inductive logic programming is therefore

performed in a *structured* hypothesis space, both to guide a search for T in the hypothesis space and to exploit properties of the chosen Q in order to determine which subspaces of the hypothesis space may be safely ignored, or *pruned*, in this search.

Inductive logic programming techniques have in common three principal ingredients. Firstly, some kind of *structure* over the space of hypotheses is adopted. Typically this structure is a preorder $\preceq$ testing whether a hypothesis is more general or specific than another. $\preceq$ is a *pre-order* over the set of hypotheses S if, for all $h \in S$, $h \preceq$ (reflexivity) and for all $h, h', h'' \in S$, if $h \preceq h'$ and $h' \preceq h''$, then $h \preceq h''$ (transitivity). Secondly, a systematic and efficient means of *searching* the space structured by $\preceq$ is adopted. While the structure defines comparability between two hypotheses, the search method typically generates a new hypothesis h' given an existing hypothesis h, such that $h \preceq h'$ under the ordering. Thirdly, appropriate means of enforcing *bounds* on the search are defined. In many cases, searches may be overly complex or the hypothesis space may be infinite. Bounds enforce stopping criteria for searches, often in terms of some syntactic restriction on the hypotheses being searched. We consider these ingredients separately in our general discussion of inductive logic programming in sections 4.1.2, 4.1.3 and 4.1.3. We then consider specific approaches for CORLOG in sections 4.2 and4.3.

The most common means to structure hypotheses is according to a generality relation. This relation compares two hypotheses in terms of the examples they cover in the dataset. We define the coverage relation $c(h, D)$ as the subset of the example set D described by the hypothesis h. Where learning is from entailment, recall that $c(h, D) = \{e \in D | h \models e\}$. Since the coverage relation is a relation over $\mathcal{L}_h \times \mathcal{L}_e$, we may extend this concept from a specific database D to the language of examples $\mathcal{L}_e$, defining $c(h, \mathcal{L}_e)$ as the set of *possible* examples in $\mathcal{L}_e$ covered by a hypothesis h. We abbreviate this so that $c(h) = c(h, \mathcal{L}_e)$. From this, we can define a generality relation in terms of c as follows:

Definition 4.6 (more general than, specialisation, generalisation, proper generalisation). A hypothesis h at least as general as a hypothesis h' for a data set D, denoted $h \succeq h'$, iff $c(h') \subseteq c(h)$. If $c(h') \subset c(h)$, we say h is more general than h', denoted $h \succ h'$.

The generality relation described is a pre-order over $\mathcal{L}$, since it is transitive and reflexive. It is, however, not a partial order, since it is not anti-symmetric, i.e., it is not the case that where $h \succeq h'$ and $h' \succeq h$, we have $h \equiv h'$, since a number of (possibly unrelated) hypotheses may cover the same examples. These hypotheses are termed *syntactic variants*.

In learning from entailment, the definition of generality in terms of the covering relation c directly is such that $h \succeq h'$ if and only if $h \wedge B \models h'$ with respect to background knowledge B. Introducing this generality ordering over hypotheses then links the logical framework of entailment to the common and well-studied notion of searching a space of hypotheses. We have seen in the previous chapter that a number of possible *syntactic* deductive operators, denoted $\vdash$, may be chosen to implement the semantic notion of entailment, denoted $\models$. These deductive operators give rise to syntactic generality orderings over the space of hypotheses. We denote these syntactic orderings $\geq$, by analogy with the semantic ordering $\succeq$, and denote by $h \geq h'$ that a hypothesis h is as least as general as h' under the ordering $\geq$. By analogy with resolution procedures, generality orderings are said to be *sound* if $h \geq h' \rightarrow h \models h'$ ($h \succeq h'$) and *complete* if $h \models h' \rightarrow h \geq h'$. In this section, we review some of the main syntactic generality orderings $\geq$ adopted in ILP systems. We introduced the concept of θ-*subsumption* [113] in chapter 3. We denote the order introduced by θ-subsumption as $\geq_\theta$.

Definition 4.7 (θ-subsumption). Recall from chapter 3 that a substitution $\theta = \{V_1/t_1, ..., V_n/t_n\}$ is an assign-

ment of terms t_i to variables V_i and that applying θ to a term, atom or clause A yields $A\theta$, in which all occurrences of V_i in A are replaced by t_i. For atoms, $a \geq_\theta a'$ if there is a θ such that $a\theta = a'$. For clauses, $h \geq_\theta h'$ if there is a θ such that $h\theta \subseteq h'$.

θ-subsumption is sound but not complete for recursive clauses. θ-subsumption tests tend to be rather computationally expensive; testing two clauses for θ-subsumption is NP-complete. θ-subsumption introduces syntactic variants. Accordingly, when working under θ-subsumption, clauses are frequently *reduced*, removing those literals subsumed by the other literals, in order to result in a *representative* of a set of equivalent clauses up to variable renaming. In [113], Plotkin shows that if for some θ and a clause C, if $C\theta \subseteq C$, there is such a reduced clause $D \subseteq C\theta$ such that C subsumes D. However, the complexity of subsumption checking is determined by the number of literals, and so reduction makes clauses more tractable.

The *object identity bias* may be imposed on θ-subsumption to obtain OI-subsumption [40], a restriction of θ-subsumption. In this, it is assumed that two distinct terms in a clause are not equal to each other. Clauses are assumed to be function-free under OI-subsumption. We denote the order introduced by OI-subsumption as $\geq_{OI}$.

Definition 4.8 (completion, OI-subsumption [40]). A function-free clause h is *completed*, denoted $com(h)$ by adding a new set of literals of the form $t \neq t'$ where t and t' are distinct terms (variables or constants) in h. A clause h *OI-subsumes* h', denoted $h \geq_{OI} h'$, if there is a substitution θ such that $com(h)\theta \subseteq com(h')$.

The extra constraints added by completion mean that if h OI-subsumes h', then h θ-subsumes h', but the converse is not necessarily true. Furthermore, the equivalence issues of θ-subsumption no longer hold; h and h' are equivalent under OI-subsumption ($h \geq_{OI} h'$ and $h' \geq_{OI} h$) only if they are variable renamings of each other. We will see in the next section that OI-substitution simplifies the search space and results in several desirable properties. However, the additional implied constraints mean that testing subsumption may be more expensive, since the clauses must be necessarily completed before the test. Furthermore, executing completed clauses is computationally more expensive than their uncompleted counterparts.

A drawback of θ-subsumption and OI-subsumption comes from the fact that they define syntactic relationships among two individual clauses and do not take further background knowledge into account, providing purely syntactic approaches to generality ordering. A number of generality orderings have extended the notion of θ-subsumption to include background knowledge. Inference rules of the form $h \vdash h'$ derive the subsumption condition in θ-subsumption and its variant OI-subsumption. Approaches taking into account background knowledge instead derive their conditions from $h \cup B \vdash h'$, for a background theory B, and allow more subsumption relationships to be taken into account. Each ordering therefore is associated with an approach to deduction. Some of the metaknowledge and elements of the object model introduced in chapter 3 can be incorporated in such a way.

We briefly review some approaches to incorporating background knowledge in generality orderings in the remainder of this section. The earliest approach to background knowledge in *subsumption* came with Plotkin's *relative subsumption* [114, 115]. We denote relative subsumption with respect to some background knowledge B as $\geq_{RS,B}$. Firstly, observe that if $h \geq_\theta h'$, then $\models \forall(h\theta \rightarrow h')$.

Definition 4.9 (relative subsumption [115]). For two clauses h and h', h subsumes h' relative to B, denoted $h \geq_{RS,B} h'$, if there is a substitution θ such that $B \models \forall(h\theta \rightarrow h')$.

Equivalently [114], $h \geq_{RS,B} h'$ if and only if there exists some h'' such that $B \models \forall(h'' \leftrightarrow h')$ and $h \geq_\theta h''$. Relative subsumption is reflexive and transitive and is therefore a pre-order. Furthermore, it is strictly stronger than θ-subsumption [111]. $h \geq_{RS,B} h'$ may also also defined in terms of deduction; it holds where there is a deduction of h' from $h \cup B$ in which h only occurs once in the derivation, as a leaf.

Buntine's *generalised subsumption* framework [13, 14] is a special case of relative subsumption. It takes a definite program B as background knowledge and applies a *model-theoretic* approach to incorporating background knowledge. We denote generalised subsumption with respect to B as $\geq_{GS,B}$. For two definite clauses h and h', h is more general than h' under generalised subsumption, denoted here $h \geq_{GS,B} h'$, if the set of atoms covered by h is a superset of the set of atoms covered by h' with respect to a Herbrand model I of the background knowledge B, where h covers an atom a under I if there is a grounding substitution θ for h such that $h\theta$'s body is true under I and $h\theta$'s head is a. More formally,

Definition 4.10 (generalised subsumption [14]). For two definite clauses h and h', h subsumes h' under generalised subsumption with respect to B (denoted $h \geq_{GS,B} h'$) if for every Herbrand model M of B and every ground atom a such that h' covers a under M, we have that h covers a under M.

Observe that if $h \geq_\theta h'$, $h \geq_{GS,B} h'$. Like relative subsumption, generalised subsumption is reflexive and transitive and is therefore a pre-order. However, relative subsumption ($\geq_{RS,B}$) is strictly stronger than generalised subsumption ($\geq_{GS,B}$). Generalised subsumption is closely related to deduction — if a clause D is a binary resolvent of two clauses standardised apart, C and E, then $C \geq_{GS,E} D$. Because this is true, $h \geq_{GS,B} h'$ iff there is an SLD-deduction of h', with h used once in the top clause and members of B as input clauses during the deduction.

Up to this point we have considered generality orderings which use subsumption. Finally, relative implication [111] uses logical *implication* instead. We denote relative implication with respect to B as $\geq_{\models,B}$.

Definition 4.11 (relative implication [111]). A clause h is more general than another clause under relative implication h', denoted $h \geq_{\models,B} h'$ if h logically implies h' relative to some background knowledge B, *i.e.* $\{h\} \cup B \models h'$. (and $B \models \forall(h) \rightarrow \forall(h')$).

Again, it is a pre-order, being reflexive and transitive. If $h \geq_{RS,B} h'$, then $h \geq_{\models,B} h'$; relative implication is a strictly stronger pre-order than relative subsumption. With respect to its relationship with induction, $h \geq_{\models,B} h'$ if there exists a deduction of h' from $\{h\} \cup B$. Relative implication characterises *implication* as a generality order. However, subsumption-based orderings are much more often used in preference to implication. Subsumption is decidable between clauses while implication is not. More efficient approaches may also be implemented using subsumption. The disadvantages of adopting a subsumption order over an implication order are that a clause may be more general than another under implication but not under subsumption. The two clauses $h = p(f(x)) \leftarrow p(x)$ and $h' = p(f(f(x)) \leftarrow p(x)$ have this property. Under implication, $h \geq_{\models,B} h'$, but under θ-subsumption, $h \not\geq_\theta h'$. These problems occur as a result of the presence of functors and relative clauses. Therefore, in databases where recursion and functors appear, implication is preferred.

Lattices from orderings

Having defined orderings over the clauses, it is necessary to impose some kind of traversable structure over them. We can use the ordering to arrange clauses into a lattice structure, from which we can define search methods with desirable properties in terms of this lattice. We discuss the traversal itself in section 4.1.3, and

introduce key concepts of lattices here. Each order reviewed above is reflexive and transitive, and therefore define a pre-order. A set equipped with a pre-order is called a pre-ordered set; the set of clauses with the pre-order of the generality relation is an example of a pre-ordered set. We identify two useful properties of pre-orders for hypothesis space search. Antisymmetric pre-orders are partial orders:

Definition 4.12 (partially ordered set). A pre-order $\geq$ on a set S, is a *partial order* on S if it is anti-symmetric (for all $x, y \in S$, $x \geq y$ and $y \geq x$ implies $x = y$). S equipped with $\geq$ is a partially-ordered set, denoted $\langle S, \geq \rangle$.

On the other hand, *symmetric* pre-orders are equivalence relations:

Definition 4.13 (equivalence relation). A pre-order $\equiv$ on a set S, $\equiv$ is an *equivalence relation* on S if it is symmetric (for all $x, y \in S$, $x \equiv y$ implies $y \equiv x$).

An equivalence relation thus partitions S into disjoint equivalence classes. Considering the partially ordered set in definition 4.12, we extend the notation to $x > y$ $(x, y \in S)$ if $x \geq y$ and $y \not\geq x$. We say x and y are *incomparable* if $x \not\geq y$ and $y \not\geq x$, and that x and y are *equivalent*, denoted $x \equiv y$ if $x \geq y$ and $y \geq x$. The relation $\equiv$ is an equivalence relation over the set S in definition 4.12. We denote by $[x]$ the set of clauses equivalent to x under $\equiv$. Given a pre-order over S, a *partial order* can then be induced over the *equivalence classes* of S. In the context of a hypothesis search, these equivalence classes are typically clauses which are syntactic variants of each other. Accordingly, in ILP we usually consider pre-orders of clauses, with a resulting partial order on the equivalence class of clauses. Lattices are a kind of partially-ordered set which are of great use in searching a hypothesis space.

Definition 4.14 (upper bound, lower bound, lattice). For a partially-ordered set $\langle S, \geq \rangle$ and a subset $S' \subseteq S$, an element x is an *upper bound* of S' if $x \geq s$ for all $s \in S'$. x is a *least upper bound* (join, or supremum) if $y \geq x$ for all upper bounds y of S'. An element x is a *lower bound* of S' if $s \geq x$ for all $s \in S'$. x is a *greatest lower bound* (meet, or infimum) if $x \geq y$ for all lower bounds y of S'. If for every pair of elements $\{x, y\} \in S$, a least upper bound $x \sqcap y$ of $\{x, y\}$ exists and a greatest lower bound $x \sqcup y$[3] of $\{x, y\}$ exists, then the partially-ordered set $\langle S, \geq \rangle$ is a lattice.

A subset S' under a *pre*-order may have multiple *least* upper bounds $lub(S')$. Where there are multiple least upper bounds, they are equivalent under the equivalence relation. Similarly, multiple *greatest* lower bounds are also equivalent. No $lub(S')$ may exist, although in practice this can be excluded by introducing a greatest element $\top$ and a least element $\bot$. This introduces a lattice over S. Lattices give us a unifying framework for reasoning about orderings over clauses in ILP. They take into account generality orderings, the equivalence of syntactic variants, and the bounds correspond to generalisations (upper bounds), specialisations (lower bounds), least generalisations (least upper bounds), and greatest specialisations (greatest upper bounds).

Before turning our attention to search, we consider an interaction between properties of a generality ordering $\geq$ and the interestingness criterion Q which enables us to safely disregard subspaces as uninteresting. This process is termed *pruning* and relies on the properties of *monotonicity* and *anti-monotonicity*.

Definition 4.15 (monotonicity,anti-monotonicity [29]). A quality criterion Q is *monotonic* if for all hypotheses $h, h' \in \mathcal{L}$, and all possible datasets D, if $h \succeq h'$ and $Q(h, D)$, then $Q(h', D)$. Q is *anti-monotonic* if for h, h', D as above, where $h \succeq h'$, if $Q(h', D)$, then $Q(h, D)$.

[3]The lub is sometimes denoted $x \vee y$ and the glb denoted $x \wedge y$. We adopt the alternative notation to avoid confusion with the logical conjunctive and disjunctive connectives.

This definition is in terms of the generality relation $\succeq$, though wherever Q is (anti)-monotonic in $\succeq$, it is for any $\geq$ such that $h \geq h' \rightarrow h \succeq h'$. A search algorithm can use this property to disregard large areas of the search space, since it follows from this definition that with respect to the database D, if Q is anti-monotonic (resp. monotonic), and a hypothesis h does not satisfy $Q(h,D)$, then none of its specialisations (resp. generalisations) will. For example, we might define Q to be true if $fr(h,D) \geq n$ for some n, where fr is the frequency of examples in D covered by h. Such a criterion is anti-monotonic and h's specialisations need not be considered. We adopt this quality criterion in the learner proposed in this thesis, but for a sample of examples from D.

4.1.3 Search: Refinement in a structured hypothesis space

Formulating ILP as a search problem over this lattice, we wish to introduce the notion of traversing the hypothesis space of clauses (*cf.* S) with respect to a generality ordering (*cf.* $\geq$). Given an element $x \in S$, we wish to define define the 'next' element in this traversal of the lattice. This is done in terms of *covers* of the element x. The notion of a cover and set of covers is defined as follows.

Definition 4.16 (cover, complete set of covers). For a pre-ordered set $\langle S, \geq \rangle$, elements $x, y \in S$, and $>$ defined as above, x is an *upward cover* of y, and y a *downward cover* of x, if $x > y$ and there is no $z \in S$ such that $x > z > y$. A set of upward covers S_u is *complete* for an element $y \in S$ if for all $z \in S$, $z > y$ implies there is a $x \in S_u$ such that $z \geq x > y$. A set of downward covers S_d is *complete* for an element $y \in S$ if for all $z \in S$, $y > z$ implies there is a $x \in S_u$ such that $y > x \geq z$. If S_u (resp. S_d) is finite we say y has a finite complete set of upward (resp. downward) covers, and the set has at least one element from each equivalence class of upward (resp. downward) covers.

From this definition we consider the notion of a set of minimal upper bounds and minimal lower bounds.

Definition 4.17 (minimal upper bounds, maximal lower bounds). A set of minimal upper bounds S'_u of $S' \subseteq S$ is *complete* for S' if for all upper bounds $z \in S$ of S there is an $x \in S'_u$ such that $z \geq x$. A set of maximal lower bounds S'_d of $S' \subseteq S$ is *complete* for S' if for all lower bounds $z \in S$, there is an $x \in S'_d$ such that $x \geq z$. If S'_u (resp S'_d) is finite, then S has a finite complete set of minimal upper (resp. maximal lower) bounds.

Traversing this lattice of clauses therefore necessitates a means of generating a set of downward (or upward) covers from a given clause. ILP is then framed as a search over this lattice, usually beginning at $\top$, the clause with an empty body. In ILP, syntactic operators are applied to a hypothesis h to obtain a set of hypotheses $h'_1, ..., h'_n$. Each h'_i is then guaranteed to be such that $h \geq h'_i$ for the generality ordering $\geq$, although there may be a h'' such that $h \geq h''$ not equal to any h'_i. Accordingly, the operators serve to approximate the downward covers of h. These operators are termed *refinement operators* — those that return a set of specialisations are termed *downward refinement operators* and those that return a set of generalisations *upward refinement operators*. Refinement operators induce a refinement graph, where there is an edge from h to h' if h' is in the set generated from h.

Refinement operators have been a part of inductive logic programming since the very early work of Shapiro [128], where they were discussed as downward refinement operators only. Since then, refinement operators have been very widely used in ILP systems. The definition of downward and upward refinement over a hypothesis space follows from [111], defined for an ordering $\geq$ over a pre-ordered set of hypotheses G with respect to a hypothesis language $\mathcal{L}$.

Definition 4.18 (downward and upward refinement operators). Let $\langle H, \geq \rangle$ be a pre-ordered set. A *downward refinement operator* for $\langle H, \geq \rangle$ is a function ρ from $\mathcal{L}$ to $2^{\mathcal{L}}$, such that $\rho(h) \subseteq \{h' | h \geq h'\}$, for every $h \in H$. An *upward refinement operator* for $\langle H, \geq \rangle$ is a function δ from $\mathcal{L}$ to $2^{\mathcal{L}}$, such that $\delta(h) \subseteq \{h' | h' \geq h\}$, for every $h \in H$.

Of interest in this thesis is downward refinement, corresponding to top-down ILP. We therefore focus on downward refinement in the remainder of this chapter. We are particularly interested in downward refinemnt operators which return the set of maximal specialisation of a clause. These are termed cover-refinement operators [30].

Definition 4.19 (cover-refinement operator). Where a refinement operator ρ has the following properties, it is called a *cover-refinement operator*. For every $h \in \mathcal{L}$, $\rho(h)$ is a set of maximal specialisations of h. $\rho^*(\top) = \mathcal{L}$ for $\top$, the top element of $\mathcal{L}$ and ρ^*, the transitive closure of ρ.

There are many properties of refinement operators which are important to consider when defining ρ for a particular generality ordering Among these, three properties are considered most important. We discuss them informally, later introducing more mathematical definitions. Firstly, in order that we may compare each generated specialisation, we wish for the set $\rho(h)$ to be finite for any h — we term this property *local finiteness*. Secondly, we wish that the refinement operator is able to reach any specialisation of a clause by a finite number of successive applications of the operator. This property is called *completeness*. If there exists at most one unique sequence of applications of ρ, the operator is *non-redundant*. Finally, to avoid cycles in which equivalent clauses are repeatedly generated, we wish each specialisation under the operator to be a proper specialisation, ie $h > h'$ rather than $h \geq h'$. This property is known as *properness*. Together, these three conditions define an *ideal* operator. Unfortunately, ideal operators are not always possible unless under the simplest of generality orders (*i.e.* over atoms only); often the operator can only be shown to be finite and complete. Indeed, it can be proven that ideal operators cannot be defined for full clausal logic or even Horn clauses under subsumption, as finite complete sets of covers do not always exist, although syntactic restrictions can be applied to allow this [111].

Since ideality is not always possible, *optimality* is often adopted as a desirable property of a refinement operator. An optimal operator is one in which the refinement graph becomes a tree in which each hypothesis appears only once. Equivalently, for each $h > h'$, there is one chain of applications of ρ through the refinement tree from h to h'. Optimality is a property which leads to efficient searches through the hypothesis space, since no hypothesis is generated more than once. Optimal operators therefore carry a notion of non-redundancy, often employing additional orderings over atoms and literals and canonical or representative forms of clauses in order to remove redundancies. Optimality is often studied in the context of cover-refinement operators. Unfortunately, for most generality pre-orders over clauses, optimal cover-refinement operators do not exist, although optimal refinement is possible over atoms. In particular, it is noteworthy that Nienhuys-Cheng and de Wolf state that optimal cover-refinement operators do not exist for any clausal language with predicate or function symbols of arity 2 or more [111]. We therefore concern ourselves mainly with the notion of reducing redundacy, within the more general setting of optimality.

The efficiency of ideal and optimal refinement operators is dependent on the density of solutions in the search space [8]. In search spaces with dense solutions, ideal refinement operators are more suitable, since almost any refinement path leads to a solution. In such a space, an optimal operator may get close to a solution but some non-redundancy measure may cause it to backtrack and miss the solution, even though the density

of the search space would mean that even an optimal operator is likely to return a solution, though perhaps not as efficiently as an ideal refinement operator. However, in spaces with rare solutions, optimal refinement operators are more suitable. In these cases, a large proportion of the search space would need to be traversed and redundancies in an ideal operator would lead to duplication of the search space and potentially highly undesirable computational running times. Unless we are dealing with a hypothesis space with a very high solution density, we prefer an optimal operator over an ideal one [8].

Later in this thesis we consider a propositionalisation setting for ILP, in which a set of candidate features are generated, transformed to attribute-value form, and a propositional learner applied. In this setting, the redundancy of refinement operators are again important, to avoid unnecessary duplication of generated features, and therefore dimensionality and issues of computational complexity.

Finally, we summarise and formalise the properties of interest of a refinement operator as follows:

Definition 4.20 (properties of refinement operators). Where ρ is a downward refinement operator for a pre-ordered set $(\mathcal{L}, \geq)$, ρ^* the transitive closure of ρ, and $\sim$ the equivalence relation introduced by $\geq$.

- ρ is *locally finite* iff for every $C \in \mathcal{L}$, $\rho(C)$ is finite and computable. ρ is *complete* iff for every $C, D \in \mathcal{L}$ such that $C > D$, there is an $E \in \rho^*(h)$ such that $D \sim E$ (D and E are equivalent in the $\geq$-order). ρ is *proper* iff for every $h \in \mathcal{L}$, $\rho(h) \subseteq \{D | C > D\}$. ρ is *ideal* iff ρ is locally finite, complete, and proper.

- A cover-refinement operator ρ is *optimal* iff for every $C, D, E \in \mathcal{L}$, $E \in \rho^*(C)$ and $E \in \rho^*(D)$ implies $C \in \rho^*(D)$ or $D \in \rho^*(C)$.

Refinement operators may therefore use background knowledge where they are derived from an ordering which takes background knowledge into account. This is not the only means of incorporating background knowledge, however. For example, PROGOL [101] considers each example as a clause and constructs a *bottom clause* $\perp(c)$, the most specific clause covering background theory B such that $B \cup \perp(c) \vdash c$. In doing so, background knowledge is transformed into a lower bound on the hypothesis space, such that any hypothesis h searched in PROGOL's top-down θ-subsumption-based search must be more general than $\perp(c)$. Another example comes from the GOLEM [105] system, in which relative least general generalisations (rlggs) are employed. An rlgg of two clauses is the least general generalisation possible from these clauses, but where the literals from the background knowledge which are provable from the literals in the clauses are included. This brings about another form of search and generality ordering. However, it is noteworthy that for generality orders other than subsumption, the existence of least generalisations is not always defined in the general case [111].

Further constraining search with search and validation bias

Refinement operators present a means to search the structured space of hypotheses. However, without further constraints on the search, this would result in an exhaustive search of the hypothesis space, which would take far too much time. Therefore, we consider further restrictions on the search space. These form part of the learning bias; recall that bias is usually characterised as one of three seperate types of constraint [110, 111]. Language bias defines restrctions on the clauses in the search space, search bias defines the way a system searches this space of clauses and validation bias concerns stopping criteria of the learner. We consider search bias and validation bias here.

Firstly, search bias may be formulated as the mechanism that the learner uses to decide which areas of the search space are of interest, and which can be ignored. Rather than generating all possible clauses in $\mathcal{L}$, in

practice, quality criteria and heuristics are often used to search the hypothesis space in a more informed way. It is here that pruning techniques may be employed to improve search efficiency. De Raedt [29] characterises pruning techniques as being either *sound* — in which the (anti-)monotonic property of Q guarantees that areas of the search space are pruned which cannot contain a solution — or *heuristic* — in which a heuristic is employed to determine whether a space is likely to contain good solutions. Sound pruning techniques are of particular relevance when combined with optimal downward refinement operators, since they ensure that a pruned hypothesis and its specialisations will never re-appear in the search.

The heuristics used in heuristic pruning are typically not (anti-)monotonic and may be employed in addition to a quality criterion. We consider their role as part of a greedy, breadth-first search. Given a starting clause h, from the start of a queue, which generates refinements $H' = \rho(h)$, any hypotheses satisfying Q in H' are output. Those hypotheses in H' which satisfy the heuristic are then added to the end of the queue, and the next starting clause in the queue is taken. Such approaches suit ideal refinement operators better than optimal ones. Since the refinements of h from an ideal operator are all those related to h under its generality order, a search employing heuristic pruning is likely to yield a more meaningful exploration of the hypothesis space. The non-redunancy of an optimal operator means that only a fraction of those children closely related to the hypothesis are considered. Indeed, if a clause is pruned heuristically under an optimal refinement operator, its refinements will never be considered.

Observe that determining Q may involve a potentially expensive coverage test over each of the examples in D. Applying a heuristic may lead to a similar problem. We may therefore identify a subset $D' \subseteq D$ of the data in order to determine whether Q or a heuristic holds over it, for example in the correctness criteria of the predictive ILP task. For example, we may adopt $Q = fr(h, D') \geq n$ for the frequency measure in the example above. Where Q is (anti-)monotonic over D it is (anti-)monotonic over D'. Where the heuristic is defined in terms of the data rather than some syntactic restriction for example, it assumes that D' is representative of D. The method of selecting thus forms part of the search bias [110], together with the choice of Q, any heuristics and the generality ordering $\geq$ itself.

Finally, validation bias consists of stopping criteria for the search. Usually in ILP, Q is set to be the condition that the hypothesis covers all of the positive examples and none of the negative examples. We may stop the search on the first h such that this Q holds, or alternatively may wish to continue the search for further hypotheses which may be more satisfactory, such as one which is less complex or which is less likely to overfit the data.

Furthermore, we may be learning from imperfect data. Following [77] and [79], we can characterise imperfect data as that which contains random errors (*noise*) in the examples or background knowledge; a sparse set of examples which do not lend themselves to the detection of regularities; imperfect background knowledge lacking useful clauses or predicates or containing irrelevant ones; and missing argument values in examples. Of these, noise is the most relevant to this discussion. Under noisy data, we may choose to relax Q to allow less than total correctness. This then corresponds to a weaker stopping criterion defined in Q.

4.2 Structuring the space of features: Constrained subsumption in CORLOG

We now turn our attention to the application of ILP to the CORLOG framework introduced in chapter 3. We adopt refinement operators for the search and consequently need to define a generality order $\geq_O$ over the space of CORLOG clauses. We expand on the notion of substitution in CORLOG (and F-Logic) introduced in chapters 2 and 3, and use them to arrive at a definition for an object-constrained variant of θ-subsumption, taking into account the object model. The graduation from the definition of the CORLOG logic to an inductive refinement operator proceeds in the following way.

Firstly, we review the notion of the space of valid clauses appearing in the language $\mathcal{L}$ in section 4.2.1. Next, a class of valid substitutions are defined to guarantee this validity in section 4.2.1. From this definition, the notion of subsumption over CORLOG clauses results and we arrive at a generality ordering $\geq_O$. Finally, we consider refinement over $\geq_O$ in section 4.3, deriving a refinement operator and discussing its properties.

4.2.1 Valid substitutions for CORLOG

In order to develop a refinement operator over the space of CORLOG clauses, we must first structure the space of clauses. We select the decidable and computationally-tractable θ-subsumption ordering. The orderings described in section 4.1.2 based the subsumption relationships on the notion of a substitution. Despite the considerable difference in syntax, we view an object logic such as CORLOG as being a restriction of logic programming, in terms of a set of additional conditions proof and model theories and valid clauses. We define a notion of substitution which respects these additional semantics. A CORLOG feature consists of two elements — the relational part and the constraint part. The notion of substitution can be generalised to both these parts. We consider each in turn.

The relational part is broadly comparable to clauses in traditional logic programming. Traditional ILP based on θ-subsumption refines a clause by adding a new literal to it, or by substituting a variable for either another variable or for a constant. This yields a new clause which necessarily is θ-subsumed by the original clause. Accordingly, we model a form of refinement in the relational part as either instantiating a method signature or by performing a substitution. However, owing to the class constraints in CORLOG, it is necessary to ensure that any substitution made is valid. That is, the resulting substitution preserves type-correctness of the feature and does not cause it to become unlinked or decomposable.

Recall the general form of a substitution $\theta = \{V_1/t_1, ..., V_n/t_n\}$, for X, a variable and t a term. Variables V carry intrinsic (and possibly multiple) class membership constraints expressed in the constraint part of the feature in the form $V : c$ stating that all substitutions for V are to be of the class c. The replaced term t_i for a variable V_i is then either a variable V_j with compatible constraints or a ground id-term o, which necessarily appears in the database and is therefore intrinsically classed. Since CORLOG features are function-free, we do not consider the possibility of variables being substituted by terms involving functors.

In a CORLOG feature, each variable may be bound by several class membership constraints, which may are denoted once for each V as $V : (c_1, ..., c_n)$ and considered as a class conjunction. Subclassing between these class membership constraints is defined as in definition 3.11. n may be bound by an optional parameter *MaxC* limiting the length of the conjunction. Recall that a ground id-term is a member of the conjunctive class $(C_1, ..., C_n)$ if it is a member of each C_i. Conjunctions only appear in class membership constraints;

the arguments of methods assume only single classes. We use the term *class* (and class membership, class constraint, etc.) to refer to both single classes ($n = 1$) in the context of classed arguments and conjunctive classes ($n \geq 1$) in the context of class membership constraints. Furthermore, we may equivalently denote a conjunctive class for a variable V in set form as $c_V = \{c_{V1}, ..., c_{Vn}\}$, $n = |c_V| \leq MaxC$.

We identify three forms of substitution in this framework:

- Variable unification. $\{X_i/X_j\}$, in which all occurrences of the variable X_i are replaced with X_j.
- Variable instantiation. $\{X_i/o\}$, in which all occurrences of the variable X_i are replaced with the constant id-term o, belonging to the same class as X_i.
- Class restriction. $\{X_i : c_i/X_i : c_i'\}$, in which the class c_i of id-terms to which X_i is bound is substituted for a (possibly conjunctive) class constraint c_i', where c_i' is a subclass of c_i. The class restriction ensures that X_i may only be bound to id-terms representing objects of the class c_i' or a subclass. This restriction allows methods to be applied to the object which may not have been possible previously.

Four aspects of the object model affect valid substitutions. Firstly, the appearance of a V in a provider argument of class C guarantees that all valid substitutions of V are of class C or a subclass of C. Where V appears in a consumer argument, the method requires that V is of class C or a subclass of C. Substitution must not violate these guarantees and expectations. Secondly, substitutions which lead to stricter class constraints must respect the class hierarchy as well as ensuring that the conditions describing interactions between provider and consumer arguments as described above still hold. Thirdly, we require that the resulting feature is linked and undecomposable. We shall see in some cases a substitution can cause these properties to no longer hold. Finally, we require the substitution to not cause any of the forms of metaknowledge described in section 3.4.1 to be violated.

4.2.2 Unification and instantiation of variables

The notion of type-safe substitutions in the presence of class constraints was introduced in section 3.3.2. This section established the conditions on a variable appearing in a CORLOG clause where its literals are covered by signatures defining moded, classed arguments. The concept of a set of permissible classes $G(V)$ for a variable was also presented. Having defined these conditions, we define the set of valid substitutions — unifications and substitutions — which preserve type-safety. Under this framework, we consider the two type of variable substitution separately, first considering unification (replacement by a variable) and then instantiation (replacement by a ground term).

Variable unification

When unifying two variables according to a substitution $\{X/X'\}$, it must be ensured that the type-safety of the feature is not violated. A valid substitution thus preserves the conditions discussed earlier regarding the interactions between the provider and consumer arguments in which the variables appear and the class consraints associated with the variables. More specifically, suppose that a variable X appears in a set of provider arguments taking classes $c_r = \{c_{r1}, ..., c_{rn}\}$, a set of consumer arguments taking classes $c_a = \{c_{a1}, ..., c_{an}\}$ and is constrained by a set of classes $c_X = \{c_{X1}, ..., c_{Xm}\}$, in order for a query on the feature to be successful, for all successful variable assigments to X, denoted $X = v$ for a ground value v, the v must necessarily be of *all*

classes c_{ri} and c_{Xi}. In order to fulfil type-correctness, each consumer class c_a must necessarily be a superclass of the class conjunction formed from c_r and c_X, which bound the variable's class.

In preparation for defining a valid class constraint for the unified variable, we define the most general common subclasses of a set of classes — the set of classes for which X must be constrained. These classes are class conjunctions. Occasionally we will refer to an argument class in the context of a class conjunction for simplicity. In defining the mgcs, we aim to derive the minimal constraint within appropriate class conjunction size bounds.

Definition 4.21 (most general common subclass). Given a set of (conjunction) classes $C = c_1, ..., c_n$, the set $mgcs(C)$, denoting the set of most general common subclasses, is the set of subclasses within some length bound N, such that

$$mgcs(C) = \{c | c \preceq C, |c| \leq N, \nexists c' \text{ such that } c \preceq c' \preceq C \text{ and } |c'| \leq N\} \tag{4.1}$$

For $\preceq$, we assume the the definition of conjunction class subclassing given in definition 3.11 and that $mgcs(C)$ is constrained to reduced forms. $mgcs(\{c_1, c_2\})$ is also denoted $c_1 \sqcup c_2$ in some literature discussing the most general common subclass, by analogy with the join in a lattice. By extension, $mgcs(C) = \bigsqcup_i c_i$.

Note that in practice, finding the most general common (conjunctive) subclass may result in a simple conjunction of the simple classes in C or may instead substitute a common subclass for a pair of classes in C. The latter case is only possible where multiple inheritance exists in the data model.

Returning to the issue of unifying the variables X and X' as described above, we define the unifiability, and the resulting constraints, of X and X' in terms of the type-safety conditions. For two variables X and X', as defined above, with provider argument classes and contraint classes $PC = \{c_{r_1}, ..., c_{r_n}, c'_{r_1}, ..., c'_{r_m}\} \cup c_X \cup c_{X'}$ (note the addition of c_X and $c_{X'}$, possibly conjunctions of classes) and consumer argument classes $CC = \{c_{a_1}, ..., c_{a_k}, c'_{a_1}, ..., c'_{a_l}\}$ X and X' are *ununifiable* if $mgcs(PC)$ is empty, *i.e.* there exists no common subclass of the provider classes PC. Thus, the unificiation is undefined. Where it is non-empty, it forms a suitable class constraint on the variable unified from X and X', being both minimal and a subclass of each of c_r, c'_r, c_X and $c_{X'}$. In practice, this mgcs may be approximated by considering the mgcs of the contraints c_X and $c_{X'}$ only, since the successful substitutions for X and X' are intrinsically bounded by the sets c_r and c'_r.

It can be seen that using an element of $mgcs(PC)$ (each element representing a possible most general common conjunctive subclass) as the class membership restriction for the unified variable necessarily results in a further restriction to the class constraint from both X and X' (and the classes of its provider arguments), which is in fact the minimal restriction which is safe to assume given the bounds on the length of a most general common subclass. As a result, where X and X' are type-correct with respect to the classes of consumer arguments, then the mgcs of their constraints will also satisfy the requirements of these arguments. That is, in the resulting feature, the class constraint of the unified variable, namely some element of $mgcs(PC)$, is a subclass of each c_{a_i}, as a result of $mgcs(PC)$ being a subclass of both $c_r \cup c_X$ and $c'_r \cup c_{X'}$. This variable unification method therefore preserves type correctness. Additionally, the class constraint for the newly-unified variable is necessarily at least as strict as for X and X'. Applying a unification thus results in a refinement. Furthermore, note that the mgcs may not necessarily be unique for a given set of classes. Accordingly, several possible unifications may be possible for a given pair of variables.

We briefly discuss how the mgcs is calculated in practice. In order to obtain a maximally-general class constraint within the N bound, we first consider the case where $N = 1$, *i.e.* only single classes are considered.

In the above unification, the class constraint on the unified variable X is any of $mgcs(PC)$. Observe that one valid unification is possible for each element of PC, since the most general class which the unified variable may take is one of the element of $mgcs(PC)$. We note that in a refinement operator, one child clause (refinement) would be produced per element of $mgcs(PC)$. Despite this increase in the complexity of the hypothesis space, multiple inheritance is relatively uncommonly used in most object models. This means that $|mgcs(PC)|$ is usually 0 (no subsumption relationship) or 1 (existence of subsumption relationship).

Where an mgcs exists for a pair of variables but their length exceeds the mgcs bound, it is sometimes possible to calculate a shorter class conjunction which, although not minimal, is a common subclass of PC. In order to reduce the length of the conjunction to a set of classes c such that $|c| \leq N$, we can find a pair of simple classes in c which themselves possess a common subclass sc, and replace them with this subclass. The length of the conjunctive class therefore is reduced. This process is informally termed class merging in this thesis. In order to indentify a pair to reduce, a heuristic is required. We attempt to be 'most general' when reducing a pair of classes to its subclass, and so we consider the loss in the number of values which a variable may take as a result of further specifying its class constraint. We define the *binding preservation* between a conjunction of two classes c and a possible reduction to a new class c', for $c' \in mgcs(c)$, as $\frac{|O_{c'}|}{|O_c|}$, for $|O_c|$ (resp. $|O_{c'}|$) the size of the set of objects of class c (resp c'). A search of possible subclasses for each pair is then carried, with the highest-preserving reductions appearing in the new class constraint. The adoption of a subclass in the unification procedure thus always guarantees that the input arguments are supplied with bindings which fulfil their requirements. In practice, this heuristic may be relaxed such that the mgcs is used where there exists at least one object in the potential merge.

Variable instantiation

As noted previously, the framework for variable unification carries over to variable instantiation, in which a variable is replaced by a ground id-term. As in the variable unification procedure, a variable may only be substituted by a constant if the introduction of the constant does not break type-safety conditions. These may be summarised in the special case of a constant as follows. For substitution substituting a variable V for a constant c in a CORLOG clause C denoted $\{X/c\}$, such that $c : c_c$ for some class in X's constraining conjunction class, the following must apply. Firstly, X may not appear as the host object of a method expression in C. We identified that each host object must appear as a variable in the language of valid clauses, and so such a substitution would be illegal. Furthermore, such substitutions are generally of little use; in general they consider the properties of only one object in the system, and are less likely to be of use in a general rule. Secondly, for each input argument to a method call where X appears, with argument class c_a, $c_c :: c_a$. The constant therefore guarantees to provide the methods in the class required by the input. Similarly, for each output argument from a method call where X appears, with argument class c_r, $c_c :: c_r$. The implicit guarantee that the output of the method provides an object of at most class c_r is similarly assured. Thirdly, the substitution should not introduce any decomposability into the clause. Finally, the substitution should not cause the clause to violate any linkage properties. The former two points are easily incorporated into a learner. The latter two points require further discussion, which takes place in the next section.

4.2.3 Linkage and decomposability

The discussion above concerns the type-safety of substitutions. However, we are also concerned with producing a useful set of testable features which are as minimal as possible. In order for the features to be testable, they must be linked. In order for features not to be expressible in terms of other features, they must be undecomposable. Decomposability and linkage properties are not usually a concern in variable *unification* — given a feature, it is not possible to make it decomposable by unifying two variables. We discuss this later. We propose that a unification of two variables X, X' in a feature which obeys the linkage constraints above will never result in a feature which no longer obeys the constraints.

Recall definition 3.6, which describes linkage properties for a feature. Each of the linkage properties can be characterised as a requirement on a variable in a given position (head output, head input, body output, body input) appearing in a given position in the rest of the feature. Since unifying a variable with another only ever causes the variable to participate in additional arguments in a feature, the linkage properties are preserved under variable unification. Specifically, considering a feature C which is linked and undecomposable, and a substitution for a variable $\theta = \{X/X'\}$ such that $C\theta = C'$, any input/output relationship is preserved under a substitution; it is impossible for an input in C' to appear without a corresponding output where it did in C, since the substitution applies to each instance of X. Likewise, it is impossible for an output in C' to appear without a corresponding input.

By contrast, variable *instantiation* may break these properties in some situations. In order to avoid redundancy in the refinement operator, it is necessary to only permit substitution in those situations where it does not cause the resulting feature to become either type-unsafe, unlinked, or decomposable. We first consider the linkage properties of a substituted feature. Linkage is preserved as a result of the requirement that we do not permit variable instantiations in the host object of a literal, that is, those corresponding to double-headed edges in the graph $VD(C)$. This ensures that every literal is linked, since a variable is never introduced as a host object unless it is previously introduced as an output from another literal, and this output/input relationship cannot be destroyed by a variable instantiation, only augmented by a variable unification. Though variable unification cannot affect the decomposability of a feature, variable instantiation can. A constant (ground id-term) represents an object independent of the influence of other arguments in the feature. Accordingly, they can lead to problems with decomposable features. In order to avoid these problems, we introduce further conditions on a substitution. Recall the notion of decomposability and the variable dependency graph from section 3.2.

Given this provider/consumer relationships between literals, we can guarantee a number of properties of a feature $C\theta$ resulting from a substitution $\theta = \{X/v\}$ given its dependency graph $VD(C)$. Specifically, we can determine the subset of variables appearing in the clause which form valid variable instantiations. Firstly, recall that the substitution must not cause any variable appearing as a host object to be grounded. Secondly, the substitution must not cause decomposability in the clause. This is the case if removing all edges with the variable X would cause the portion of the graph connecting body literals to become disconnected. Where this is true, the literals necessarily form subsets whose conjunctions are valid features themselves. In the terminology of graph theory, the edges labelled with X form a *cut set* (also vertex cut or separating set) in the graph $VD(C)$. Two forms of decomposability result, depending on the presence of the variables with underlined edge labels. If a subgraph resulting from a disconnection like this has *each* variable with a circle present in it, then it forms a valid clause with respect to linkage. Where this does not apply, literals take in disconnected subgraphs fall into three additional categories. Firstly, they may consume the variable in the host object of the head literal,

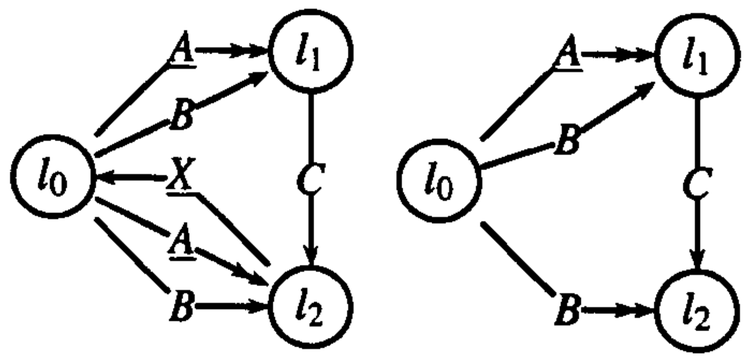


Figure 4.1: The variable dependency graphs of features made decomposable by $\{C/c\}$

in which case they require a literal returning the output object of the head literal to form a valid feature. For brevity, we term this a *head-host-consuming* subgraph (or set of literals). Secondly, they may return the variable in the output argument of the head literal, in which case the converse is true. For brevity, we term this a *head-output-providing* subgraph (or set of literals). A head-host-consuming subgraph may form a feature with a head-output-providing subgraph. Finally, neither of the above are true, in which case the literal is necessarily linked to input variables in the head literal. The double-headed edge thus denotes a variable which may not be substituted and the underlined edge labels a variable which must exist in a connected subgraph representing a linked feature.

Recall the examples of variable dependency graphs given in section 3.2. We consider the variable dependency graphs of two further CORLOG features in figure 4.1, demonstrating the effects of using the variables in a candidate substitution These variables tag edges in the graphs. The graph on the left represents the clause

$$F_1 = \mathsf{A}[\mathsf{f}_0(\mathsf{B}) \rightarrow \mathsf{X}] \leftarrow \mathsf{A}[\mathsf{f}_1(\mathsf{B}) \rightarrow \mathsf{C}], \mathsf{A}[\mathsf{f}_2(\mathsf{C}) \rightarrow \mathsf{X}].$$

and the graph on the right the clause

$$F_2 = \mathsf{A}[\mathsf{f}_0(\mathsf{B}) \rightarrow \mathsf{x}] \leftarrow \mathsf{A}[\mathsf{f}_1(\mathsf{B}) \rightarrow \mathsf{C}], \mathsf{B}[\mathsf{f}_2(\mathsf{C}) \rightarrow \mathsf{d}].$$

We study the effect of the substitution $\theta = \{C/c\}$ on these features and their decomposability.

$$\begin{aligned} F_1\theta &= \mathsf{A}[\mathsf{f}_0(\mathsf{B}) \rightarrow \mathsf{x}] \leftarrow \mathsf{A}[\mathsf{f}_1(\mathsf{B}) \rightarrow \mathsf{c}], \mathsf{A}[\mathsf{f}_2(\mathsf{c}) \rightarrow \mathsf{X}]. \\ F_2\theta &= \mathsf{A}[\mathsf{f}_0(\mathsf{B}) \rightarrow \mathsf{x}] \leftarrow \mathsf{A}[\mathsf{f}_1(\mathsf{B}) \rightarrow \mathsf{c}], \mathsf{B}[\mathsf{f}_2(\mathsf{c}) \rightarrow \mathsf{d}]. \end{aligned}$$

Since θ has the effect of cutting the edge associated with C in the graphs, we can observe that θ is not legal. In the case of F_1, the literal l_1 is no longer required to form a valid clause. Removing l_1 from $F_1\theta$ would still be valid, but as l_1 cannot act as a valid clause by itself, it is not a decomposable feature. In the case of $F_2\theta$, $l_0 \leftarrow l_1$ forms a valid clause, but l_2 does not (it would if A had been its host object). Such substitutions, then, do *not* lead to decomposability.

We now consider two that do. Figure 4.2 shows the graphs of features F_3 (left) and F_4 (right). The features are defined as follows:

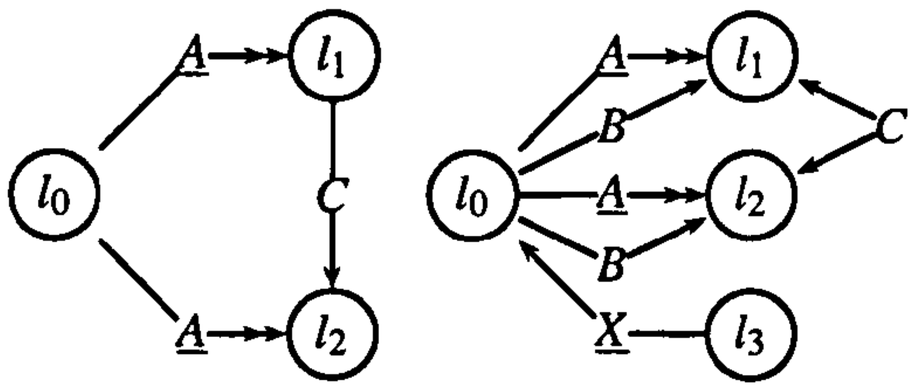


Figure 4.2: Simple examples of decomposition in variable dependency graphs

$$F_3 = \mathsf{A}[\mathsf{f}_0 \rightarrow \mathsf{x}] \leftarrow \mathsf{A}[\mathsf{f}_1 \rightarrow \mathsf{C}], \mathsf{A}[\mathsf{f}_2(\mathsf{C}) \rightarrow \mathsf{d}].$$
$$F_4 = \mathsf{A}[\mathsf{f}_0(\mathsf{B}) \rightarrow \mathsf{X}] \leftarrow \mathsf{A}[\mathsf{f}_1(\mathsf{B}) \rightarrow \mathsf{C}], \mathsf{A}[\mathsf{f}_2(\mathsf{B}) \rightarrow \mathsf{C}], \mathsf{B}[\mathsf{f}_3 \rightarrow \mathsf{X}].$$

Observe that F_4 uses an variable as the output argument of its head literal. Adopting the substitution $\theta = \{C/c\}$ as before,

$$F_3\theta = \mathsf{A}[\mathsf{f}_0 \rightarrow \mathsf{x}] \leftarrow \mathsf{A}[\mathsf{f}_1 \rightarrow \mathsf{c}], \mathsf{A}[\mathsf{f}_2(\mathsf{c}) \rightarrow \mathsf{d}].$$
$$F_4\theta = \mathsf{A}[\mathsf{f}_0(\mathsf{B}) \rightarrow \mathsf{X}] \leftarrow \mathsf{A}[\mathsf{f}_1(\mathsf{B}) \rightarrow \mathsf{c}], \mathsf{A}[\mathsf{f}_2(\mathsf{B}) \rightarrow \mathsf{c}], \mathsf{B}[\mathsf{f}_3 \rightarrow \mathsf{X}].$$

By applying the substitution θ, F_3 has been decomposed — there exist features F_{3a} and F_{3b} such that $head(F_{3a}) = head(F_{3b}) = l_0$, $body(F_{3a}) = l_1$ and $body(F_{3b}) = l_2$, where F_3 can be expressed as a conjunction of the bodies of F_{3a} and F_{3b}. F_4 has been similarly decomposed — there exist features F_{4a} and F_{4b} such that $head(F_{4a}) = head(F_{4b}) = l_0$, $body(F_{4a}) = l_1, l_3$, $body(F_{4b}) = l_2, l_3$, where F_4 can be expressed as a conjunction of the bodies of F_{4a} and F_{4b}. Observe that there are no other literals in the F_3 (resp. F_4). Were there head-host-consuming or head-output-providing literals, these could have been included in either F_{3a} or F_{3b} (resp. F_{4a} or F_{4b}) to provide a decomposition. Observe also that the decomposition in F_3 has arisen from the breaking of a input/output link (the variable C) in the clause. By contrast, F_4 was decomposed by breaking an output/output link (again the variable C). More generally, the four kinds of subgraph introduced earlier (valid clause, head-host-consuming, head-output-providing, and none of these), can work together to provide a decomposable CORLOG clause. If a decomposition of a clause C is defined as a set partition $L(C) = \dot{\bigcup} L_i$, *i.e.* $L_i \cap L_j = \emptyset \forall i, j$, such that each L_i is either:

- The literals appearing in a subgraph defining a valid clause, optionally including the literals appearing in any other kinds of subgraphs.
- The literals appearing in a subgraph defining a head-host-consuming subgraph, together with the literals appearing in a subgraph defining a head-output-consuming subgraph, optionally including the literals from any non-valid clause subgraphs.

Note that this may lead to more than one possible decomposition. Where this property holds for $L(C)$, the clause C is necessarily decomposable.

We restrict CORLOG features to simple CORLOG features. Recall from definition 3.4 that this restriction involves only a host object argument in the head, with an output argument which is always constant. This restriction is convenient with respect to undecomposability since it allows us to eliminate decomposability by requiring that the host object variable in the head is only used once in the body of the feature. While this means that a feature cannot use the host object variable for an input argument, it also necessarily removes the possibility of decomposable features. With respect to linkage, we make an additional choice in the light of the use of simple CORLOG features to only substitute those variables which are not yet consumed. This necessarily cannot produced unlinked features.

4.2.4 The class hierarchy and class metaknowledge

So far we have considered substituting variables for other variables or ground terms in the relational part of the clause, and their effect on the class membership constraints in the constraint part. In this section we discuss *class restriction* substitutions, *i.e.* those which substitute a class membership constraint $X : c$ for a variables with a stricter constraint $X : c'$, where c' is a subclass of c. This form of class restriction is also known as a *downcast*, since it is analogous to the casting operation in object-oriented programming in which one class is converted into another. In CORLOG, we consider a class to be a conjunction of classes, some of which may be parametric.

Recall the notions of an object belonging to a class conjunction from section 3.3 and the subclassing relationship among conjunctive classes from definition 3.11. This framework has a number of notable simple properties. Firstly, making one of the classes in the conjunction more specific makes the while conjunction more specific, assuming that the specified class does not have a superclass existing in the conjunction already. Secondly, we can make a class conjunction more specific by adding a new class which does not already have a superclass in the conjunction. Finally, if we denote the reduced form of a conjunctive class c as $red(c)$, the set of equivalence classes of conjunctions defined by $c \sim c'$ if $red(c) = red(c')$, form a partial order $\leq_c$ for the subclass relation, where $c \leq_c c'$ if c is a subclass of c' according to definition 3.11.

The definition of a legal substitution for a class conjunction follows naturally from that of single classes; the class restriction substitution $\{X : c/X : c'\}$ is only legal where $c' \leq_c c$. With respect to refinement, a class conjunction c is transformed into a subclass c' by either downclassing one of the elements of c or by adding another class to the conjunction (where the constraint allows it). As well as considering the coexistence conditions on classes, we discuss valid substitutions in terms of these two elementary operations. Additionally, we will discuss the concept of a minimal class restriction, which is of interest in refinement. A minimal class restriction is a valid substitution $\theta = \{X : c/X : c'\}$ for class conjunctions c and c' such that there is no class conjunction c'' for which $\{X : c/X : c''\}$ and $\{X : c''/X : c'\}$.

Observe also that the space of possible class conjunctions could be potentially very large, creating a resultingly large hypothesis space. As a result, we adopt bounds on the conjunctions which can be constructed by class substitution. These bounds are the number of classes involved in the conjunction (its size or length) and the support of the class conjunction in the database, *i.e.* the number of objects which are instances of the class conjunction. By analogy with refinement operator, we can also introduce a depth bound on the number of class restrictions which can be made to a variable.

The semantics of CORLOG introduced in chapter 2 introduce a number of restrictions on what can be considered a valid class restriction substitution. We identify these as the type-correctness restrictions arising

from the moding and classing of arguments, the semantics of parameteric classes and the bounds on their parameters and the presence of class-specific metaknowledge which must be respected in a class constraint.

Previously in this section, we considered the type-safety of a set of relational method expressions and in particular the interplay between the guarantees introduced by arguments providing variables (*e.g.* outputs of methods in the body of the feature) and the requirements of consumer arguments (*e.g.* inputs and host object arguments in the body of the feature). Specialising a class conjunction which constrains a variable always introduces a stricter class membership requirement on the successful substitutions for that variable. The type-correctness requirements of consumer arguments, which require that the terms appearing in them are of its class or a subclass, are therefore not affected by a class restriction.

The presence of parametric classes in the language of CORLOG introduces two new considerations with respect to the validity of substitutions. These are the subclassing framework and the presence of bound parameters in the definition of the parametric class. Parameteric classes were discussed in section 3.3.1. We briefly consider the handling of parametric class constraints during substitution. Recall that in the CORLOG framework, parametric classes take either exact or inheriting parameters. This aims to reflect whether a method existing for parametric class has some dependency on the bound element such as expecting it to provide a method.

In our logical framework, parametric classes are considered in the same way as simple classes, since once they are bound, they adopt the same behaviour as any other class in the system. F-Logic considers parametric classes in the same way as simple classes, placing no special semantics on them. It is therefore the responsibility of the user to define the form of the inheritance. Parametric classes are presented using functors and subclass assertions involving variables are defined within the deductive system to reflect parametric subclass. Our *induction* method assumes specific properties of parametric classes operating under two rules. Firstly, a parametric class $p\langle c_1,...,c_n\rangle$ can be defined as a subclass of another parametric class $p'\langle d_1,...,d_m\rangle$. The inheritance relationship is specified such that parameters may be mapped from p to p' via variables, or new ones may be introduced in the subclass via constants. Secondly, each parameter is defined to be inheriting or exact. A parametric class $p\langle c_1,...,c_i,...,c_n\rangle$ is a subclass of another parametric class $p\langle c_1,...,c_j,...c_n\rangle$ for two classes c_i and c_j appearing in a parameter position which is inheriting, and c_i is a subclass of c_j. Togther these rules define a full class hierarchy for each parametric class and its set of legal parameters. These are adopted as the basis for class restrictions in the inductive system.

Finally, the existence of class metaknowledge also affects the set of legal class restriction substitutions possible. Consider the candidate substitution $\theta = \{X : c/X : c'\}$. Recall that the class metaknowledge consists of declaring classes abstract or mutually disjoint and that a class conjunction is specialised by adding a new class to it or by downcasting an existing one. In applying a class restriction, we must ensure that the resulting conjunction is still valid with respect to this metaknowledge. Firstly, a class declared abstract may not appear in the class conjunction c'. A class restriction resulting in a class conjunction containing an abstract class is not valid. Instead the set of minimal class restrictions are the most general subclasses of the abstract class, which are not themselves abstract. Secondly, no pair of classes (c_i, c_j) $(c_i \in c', c_j \in c', c_i \neq c_j)$ may exist in the conjunction such that c_i and c_j, or any of their subclasses, are declared mutually-disjoint in a disjointness declaration. Finally, we require that class restrictions are non-redundant with respect to the class hierarchy. The restriction is redundant where c' is not reduced. Equivalently, it is redundant if there is a pair of classes as above such that c_i is a superclass of c_j.

4.3 Searching through the space of features: refinement in CORLOG

The previous section discussed the impact of the object-oriented data model proposed in the previous chapter on the core notions of substitution and subsumption in ILP. Specifically, we considered a number of aspects of the adopted data model and arrived at the notion of a valid substitution which respected the restrictions introduced by these aspects on a feature. We considered type-safety restrictions, determined by the interaction of input, output and host object arguments in the feature and declarations in associated method signatures augmented with meta-knowledge to arrive at the notion of a type-safe substitution for variable unification and instantiation. We discussed how to avoid decomposability in a constructed feature, and how aspects of the class metaknowledge affect class restriction substitutions. Unification and substitution may be seen as the building blocks of a refinement operator. Next, we consider the structure of the hypothesis space in terms of the whole language, ordering it with a subsumption ordering $\geq_O$ and devise methods which allow efficient search of this ordering, namely the refinement operator.

The notion of θ-subsumption, in which $h \leq_\theta h'$ if there is a θ such that $h\theta \subseteq h'$, can be adapted to the basis of CORLOG subsumption by simply restricting the set of valid substitutions as described above. A number of further considerations still remain, primarily the aspects not covered by the substitution mechanism such as argument refinement. We consider these aspects in our discussion on refinement which follows. Having structured the space of features with an ordering in this way, we consider methods of traversing the space. This is done through means of a refinement operator introduced in section 4.1.3. We discuss refinement as a process in general and arrive at a definition of the process of refinement in the space of object queries.

The notion of a refinement operator was discussed earlier in section 4.1.3. Recall that we introduced a downward refinement operator as a function ρ from $\mathcal{L}$, a hypothesis space, to $2^{\mathcal{L}}$, the power set of $\mathcal{L}$, such that for every clause $C \in G$, where G is a pre-ordered set of clauses, $\rho(C) \subseteq \{D|C \geq D\}$, that is each D is at least as specific as C. We discussed some desirable aspects of refinement operators, namely local finiteness, completeness and properness. Optimality, a property of particular interest in propositionalisation, was also discussed. We build on the concepts of a legal substitution above and the resulting notion of subsumption in order to arrive at a refinement operator for the object logic CORLOG.

In traditional ILP, the generality ordering of θ-subsumption between two clauses C and D is defined in terms of a subset criterion and a substitution θ. C θ-subsumes D (C is more general than D) if there exists a θ such that $D\theta \subseteq C$. Downward refinement, *i.e.* constructing specialisations of clauses, therefore consists of two operations derived from this formula. The subset relation leads us to add a literal to a clause to refine it, and the substitution in the formula leads us to perform a substitution in order to refine it. Defining a suitable operator, however, requires a number of new considerations in order for it to be useful in a propositionalisation learner, principally the avoidance of redundancy. This is related to the study of optimal refinement operators. Informally, an optimal refinement operator is one in which each clause in the graph has only one path from the most general clause. Non-optimality of a refinement operator comes about as a result of redundancies in clauses introduced by the application of the operator. Designing an optimal refinement operator therefore requires careful consideration so that syntactic variants of the same clause do not appear more than one in the tree of clauses induced by successive refinements. We reported in section 4.1.3, optimal refinement operators do not exist in many clausal languages and for predicates of greater than arity 2. Accordingly, decisions are made so that a feature is refined from only one parent where possible. This results in the refinement graph taking on a tree-like structure.

A number of approaches have been taken in this regard. Badea and Stanciu [8] attained an optimal refinement operator by first weakening the subsumption ordering. As in Progol, a bottom clause $\perp$ is specified, each consisting of moded literal 'templates' using variables. C weakly subsumes D if $C\theta \subseteq D$ for some θ which does not unify any literals and for which $\theta_\perp(D)\theta = \theta_\perp(C)$. Weak subsumption is stronger than the OI-subsumption discussed earlier but weaker than θ-subsumption. Each literal in $\perp$ is selected for inclusion only once. This measure has a number of useful effects, among them the guarantee that there are no infinite ascending or descending chains, the obstruction to an optimal refinement operator under θ-subsumption. It is also no longer the case that a clause with more literals can be more general than one with fewer. Weak subsumption enables the incorporation of a number of measures which ensure the optimality of a refinement operator employing them. Firstly, the operator makes use of an ordering over literals. Each subsequent addition of a literal is then defined as valid or invalid depending on the presence of existing literals and their position in the order — a literal is not added where an earlier literal in the ordering exists in the clause, unless it could not have been added previously. Furthermore, in order to enable optimality, the literals are moded, and the introduction of new variables introduces an ordering over them. Newly-introduced variables from the addition of a literal are marked as 'fresh', and may only be used in subsequent substitutions. Finally, the introduction of fresh variables may permit an earlier literal (in the ordering) to be applied where it was not possible before. This is known as 'waking' a literal, and together with adding a literal and applying a substitution to a fresh variable, completes the three possible refinement operations under the weak subsumption ordering. The refinement operator's optimality thus relies on a ordering over literals and variables, the 'freshness' of variables (whether they are newly-introduced) and a moded bottom clause whose arguments have variables acting as a template.

The optimality properties of this refinement operator can be adapted to one which operates less redundantly in the CORLOG framework. At its most broad, the refinement is the same as for θ-substitution — adding a literal or performing a substitution. Substitutions must be valid according to the criteria described above. In the framework proposed, the object-orientation of the the ILP approach exists mainly in the restricted form of the substitution, although addition of a literal must satisfy linkage properties.

4.3.1 Constituent refinement operators

In more depth, we define the refinement operator acting on CORLOG clauses as a function ρ described above. Given a clause C, the set of refinements $\rho(C)$ includes clauses produced from a set of basic operators. Each operation has a number of specific variants, each denoted by a separate ρ-function. Then, $\rho(C) = \rho_{al} \cup \rho_{wl} \cup \rho_{su} \cup \rho_{si} \cup \rho_{cr} \cup \rho_{mv} \cup \rho_{mc}$, as detailed in the list below:

- *Adding a literal.* ρ_{al} adds a new literal obeying literal ordering constraints, whereas ρ_{wl} "wakes" a literal, *i.e.* adds a literal which is ordered previously to one already appearing in the clause but whose addition was not possible before due to appropriate variables not being available. Adding a literal necessarily involves a number of substitutions in order to fill its arguments with variables linking them to the rest of the clause.

- *Performing a substitution.* Substitutions must be valid according to the criteria described above regarding type-correctness and the existence of a mgcs. ρ_{su} performs a substitution which unifies two variables whereas ρ_{si} instantiates a variable with a constant.

- *Performing a class restriction on a variable.* ρ_{cr} specialises some restriction on the class of a variable,

as defined in the valid class substitutions above. Note that class restrictions may involve simple or conjunctive classes, possibly incorporating parametric classes. Again, class restrictions fall under the framework of a valid substitution as above, and are therefore considered within the framework of the substitution itself. In order to overcome the large amount of redundancy which this constituent operator might introduce, we describe a method of optimally searching the space of legal class conjunctions.

- *Argument specialisation.* Argument specialisation is additional to the substitution framework. Where an ordering is defined using the projections introduced in the previous chapter, argument specialisation takes advantage of these orderings to specialise the clause. A variable may be replaced by the constant k_0 representing the most general restriction on the possible individuals. Subsequent refinements then refine k_0 into k_1, k_2 and so on, representing ever more strict conditions on the head object. In the first instance, argument specialisation replaces a constant for a variable. In the second, a constant (and furthermore a particular instance of a constant) is necessarily replaced by another constant. Argument specialisation is carried out by the operators ρ_{mv} on variables and ρ_{mc} on constants. Argument specialisation thus cannot be considered a form of substitution, which acts at the variable level, but instead a form of refining transformation acting at the literal level, framing the literal as a kind of simple condition or constraint such as $X < 2$.

We adopt the ILP approach of propositionalisation in this thesis, and therefore aim to remove duplication in the features produced during refinement. Optimal, or near-optimal, refinement operators are therefore of interest. A number of specific measures are necessary to avoid the duplication of equivalent clauses at different points on the refinement lattice.

We define each of the operations of the presented refinement operator in turn, drawing heavily on the notion of a valid substitution as described above. We adopt an example from the ILP mutagenesis domain, to illustrate each one. For simplicity and brevity, the method calls in these examples rarely take any arguments. However, it should be noted that refinement over method calls for arbitrary arity are defined. To clarify the effect of each type of refinement, we adopt the notation $\underline{\mathsf{X}}$ to indicate where an element X (variable, method or constant) of a clause has been changed.

In each case we refer to a clause D for which $D \in \rho(C)$. Each D must be produced as a result of the application of one of these operators. The variables of the clause C are denoted by $Vars(C)$. For each refinement operator, a number of conditions on its application is given. Following [8], an (arbitrary) ordering is imposed on the set of possible literal symbols, or in the CORLOG nomenclature, the method signatures. Furthermore, the variables are ordered by their introduction during refinement and a set F of 'fresh' variables maintained, representing the variables which may be substituted during refinement. In our learner, it is possible to reuse variables which are no longer fresh. The result is an increase in expressiveness at the cost of redudancy. For the examples given below, this setting is adopted for simplicity.

Adding a literal — ρ_{al} and ρ_{wl}

$D = C \cup L'$ where L' is a molecule $\mathsf{O[M(A_1, A_2, ..., A_n) \rightarrow R]}$ obtained by instantiating a signature molecule $S = \mathsf{C_O[M(C_{A_1}, C_{A_2}, ..., C_{A_n}) \Rightarrow C_R]}$.

The following conditions must hold for the operation ρ_{al}:

1. L' conforms to the signature molecule S.

2. L' is ordered later (in the literal ordering) than each of the literals in C.

3. The variable of the host object O is a fresh variable in F. Since variables may be downcasting, the application of the method should occur at the earliest possible opportunity and no later. As a result, V must either be (i) a variable which has not yet been downcasted or (ii) a variable for which a type specification operation has recently caused the class of the variable to be such that the method is applicable to it. Furthermore, where the class of the method C_O is a variablised parametric class, the method signature must be made concrete with respect to the variable being invoked on it, and the resulting concrete signature adopted for the remainder of the literal application.

4. The variables of each of the input arguments A_i are variables existing in C, in the set $Vars(C)$. Necessarily, this involves an intrinsic substitution and that the variables must be unifiable according to the substitution framework in section 4.2.2, since class constraints are introduced by those already existing in the clause and the method signature being instantiated. The operation is thus equivalent to first instantiating the method signature with new variable constrained according to the classes C_{A_i} and then performing variable unification substitutions on the variables O and each A_i according to the substitution constraints. Where the method is also order-yielding, the special case in which the input variable introduces a new (fresh) variable (for later refinement), also holds. Like the output variable, the new input variable takes the type of its argument.

5. The variable of the output argument R is a new and distinct variable in the clause, with class C_R. The new variables are ordered in such a way that each output variable in C precedes that of those in L'. As a result, $Vars(D) = Vars(C) \cup R$, and a new class membership constraint is introduced into the CORLOG clause to reflect the method signature's type-correctness requirements on the variable.

6. F, the set of fresh variables, is set to the new output variables introduced — in our framework, R.

7. The literal L' may not already exist in C.

The refinement operator ρ_{wl} is a variant of ρ_{al} which enables the subsequent addition of a literal where it was not possible before due to the lack of required variables in the (partially-refined) clause. ρ_{wl} is as for ρ_{al} in terms of variable substitution and the selection of fresh variables. However, the following differences apply:

1. L' is based on a method signature ordered earlier than the lastest in C, but which has not yet been used in C.

2. For all literals $L_i \in C$ such that $L_i > L'$ in the literal ordering, if $first_{L_i}$ are the literals added before L_i, then the clause $first_{L_i} \cup \{L'\}$ is invalid.

For example, the refinement by literal addition of the query

$$\mathsf{M[mutagenic \rightarrow yes] \leftarrow M : molecule} \tag{4.2}$$

given a signature

$$\mathsf{molecule[hasgroup \Rightarrow group]} \tag{4.3}$$

includes

$$\mathsf{M[mutagenic \rightarrow yes] \leftarrow \underline{M[hasgroup \rightarrow G]}, M : molecule, \underline{G : group}} \tag{4.4}$$

Performing substitutions — ρ_{su} and ρ_{si}

Having added literals to a clause, they can be further refined by applying substitutions. We distinguish between two kinds of substitution — those which replace a variable by another variable (unification, using ρ_{su}) and those which replace a variable by a constant (instantiation, using ρ_{si}).

Variable unification (ρ_{su}). $D = C\theta$ for a substitution $\theta = \{x/t\}$ where a term t is substituted for a distinct variable x. Where ρ_{su} is applied, both x and t are variables and exist in $Vars(C)$. For optimality, the following restrictions are placed on the applicability of a substitution:

1. t and x must be unifiable according to the requirements for a valid variable unification substitution (mgcs) in section 4.2.2.
2. t must be before x in the variable ordering.
3. x must be a fresh variable.
4. F is reset to include only the variables which succeed x in the ordering from the original F.

For example, the refinement using variable unification of the query

$$\mathsf{M[mutagenic \rightarrow yes] \leftarrow M[hasgroup \rightarrow G1], M[hasgroup \rightarrow G2], G1[numatoms \rightarrow N1], G2[numatoms \rightarrow N2],}$$
$$\mathsf{M : molecule, G1 : group, G2 : group, N1 : integer, N2 : integer} \quad (4.5)$$

includes

$$\mathsf{M[mutagenic \rightarrow yes] \leftarrow M[hasgroup \rightarrow G1], M[hasgroup \rightarrow G2], G1[numatoms \rightarrow N1], G2[numatoms \rightarrow \underline{N1}],}$$
$$\mathsf{M : molecule, G1 : group, G2 : group.N1 : integer} \quad (4.6)$$

Variable instantiation (ρ_{si}). We adapt the notion of a variable unification substitution directly to a variable instantiation, adopting the requirements for a valid variable instantiation instead of those for unification. Before doing so, we distinguish between methods which are order-yielding and those that are not. Substitutions involving variables occupying these arguments are referred to as *ordered substitutions*, and if not, as *non-ordered substitutions*. ρ_{si} performs non-ordered substitutions, and therefore covers substitutions for arguments which have no concept of order. Later we discuss ordered substitutions.

Typically, there will exist classes to which a large number of object will belong. For example, a database describing many people may have thousands of person objects. Substituting each of these objects for a variable of class person would produce one feature for each object, leading to a large number of features which are unlikely to be useful for building a rule.

Only particular classes are considered to be valid for variable instantiation. The class must be declared as *subprimitive*. These classes are necessarily a subclass of a class to which (constant) atoms and numbers belong, namely the symbol, integer, number and float classes. Additionally, the class must have no more than the number of member objects defined by the parameter `largetype`. Unless these conditions are met, a variable may not be substituted by a constant by the refinement operator presented.

The conditions on a valid substitution $\{x/t\}$ under ρ_{si} are as follows:

1. t and x must be unifiable according to the requirements for a valid variable instantiation.

2. x must be a fresh variable.

3. After substitution, F is re-set to include only the variables which succeed x in the ordering from the original F.

For example, the refinement using variable instantiation of the query

$$\mathsf{M[mutagenic \rightarrow yes] \leftarrow M[hasgroup \rightarrow G], G[atomcount \rightarrow A], M : molecule, G : group, A : integer} \quad (4.7)$$

includes

$$\mathsf{M[mutagenic \rightarrow yes] \leftarrow M[hasgroup \rightarrow G], G[atomcount \rightarrow \underline{10}], M : molecule, G : group} \quad (4.8)$$

Performing a class restriction — ρ_{cr}

By *arbitrarily* downcasting elements of the class conjunction and adding new classes, the resulting search defined by these refinements is highly suboptimal. Since we take conjunctions of classes, the resulting space being searched may be infeasibly large, especially where parametric classes are used. CoSinus uses an optimal type specialisation operator which operates according the class metaknowledge restrictions of abstract classes, mutually-disjoint classes and parametric classes. We use the notion of class membership restriction to arrive at a refinement operator which produces class restrictions on clauses whilst maintaining optimality in this refinement operator responsible for class specialisation. We may restrict a class conjunction by adding a class to it, downclassing an existing class, or removing a class and replacing it with a conjunction of (a subset of) its subclasses. In isolation, the class specialisation operate can be seen as a producing a graph consisting of class conjunctions, analogous to the clause refinement graph, in which an edge exists from a class conjunction c to another c' if c subsumes c'. For an optimal refinement operator, this lattice instead takes the form of a tree, and therefore each class conjunction has exactly one path from the most general class conjunction (top). As with the operations which add literals and perform substitutions on clauses, it is necessary to introduce a number of measures to ensure that for any given class structures, this property holds.

In order to apply the algorithm, it is necessary to impose a wellorder $\leq_C$ over all possible *simple* classes (and parameterisations of classes) C, by establishing a topologically-sorted list $\langle c_1, ..., c_n \rangle$ of classes, assigning indices to them ($I(c_i) = i$), such that if c_i is a superclass of c_j, $i < j$. This wellorder is used to define a spanning tree over the hierarchy of classes. Each class which has a superclass has a *first parent*, the parent with the minimum index of all its parents. Where a superclass is declared abstract, it is ignored in the class hierarchy, and its *superclass* is instead considered as a first parent. We make extensive use of this concept and this is the principle mechanism by which abstract classes are accommodated. In order to ensure optimality, we introduce additional values associated with a class conjunction at an arbitrary point in the refinement tree. In establishing these concepts, we model the class conjunction as an sequence of classes. The first value is the *maxorder M*, which is updated only when a new class is added to the conjunction with the index of the new class added. The second value is a list of *fresh positions P*. This value changes during a downcast, and reflects the positions

in the sequence in which further downcasts can be made. Further additions will reset this list to *any*, so that downcasts can be made in any position. We represent these additional tagged values by extending the notion of class constraint $(V : C)$ to a 4-tuple $\langle V, C, M, P \rangle$, tagging a variable V to C, a *sequence* of classes, M is the maxorder number, and P is the set of positions in the sequence which are fresh. A new variable V of class C is created with the 4-tuple $\langle V, C, -1, \{1, ..., |C|\} \rangle$. Allowing *any* possible combination of classes is unlikely to lead to general rules and causes the space of possible conjunctions to increase exponentially. We therefore limit the size of a conjunction with the *MaxCC* bound, in which any conjunction $C = \{c_1, ..., c_n\}$ considered must be such that $|C| \leq MaxCC$. The depth of the downcasting may also be limited. There are three such bounds. Further bounds limit the number of downcasting and addition operations possible on a new variable. The maximum number of operations is limited by *MaxDC*, with *MaxDCS* and *MaxDCP* further bounding the search on simple and parametric subclasses.

$D \in \rho_{cr}(C)$ if, for some fresh variable $X \in Vars(C)$ with class conjunction c is replaced by the variable $X \in Vars(D)$ with a new class conjunction c' where c' results from either a class being added to c or one of the elements of c being type specialised. We consider the conditions on each operation, where $\langle X, c, M, P \rangle$ is the extended constraint on X prior to refinement and $\langle X, c', M', P' \rangle$ is the constraint after refinement.

Firstly we consider the addition of a class t to a conjunction c, such that $c' = c \cup t$ for some class c in the database. Firstly, t *must qualify for addition.* In order to 'qualify', t must have no superclasses in the conjunction c (in which case its addition would make the superclass redundant) and it must have no subclasses in c (in which case it is covered later in further refinements). Further, it must not cause c' to break any mutually-disjoint sets. Secondly, *the first parent of t must not qualify for addition.* t must be the most general class which qualifies, in order that all possible class combinations are considered. Thirdly, *c' must be a metaknowledge-valid combination.* c' must not contain any abstract classes, must not contain any pair of classes which are from the same mutually-disjoint set, must not contain any pair of classes (r', r'') such that t' is a subclass of r'', and there must be an object in the database of (conjunctive class) t'. Fourthly, *c must be in order.* $I(c) > M$. Then, $M' = I(c)$ and $P' = \{1, ..., |C|\}$. Finally, *refinement bounds must be respected.* $|C| < MaxCC$, and no more than *MaxDC* add or downcasting operations have been performed on the conjunction og which no more than *MaxDCS* have been on simple class and no more than *MaxDCP* have been on parametric classes. These are counts carried with the conjunction but are ommitted from it for clarity.

Secondly we consider the downcasting of a class in a conjunction such that $c' = c \backslash t \cup t'$ for classes t and t' in the database. Firstly, *the position of t in c is a valid position in P.* Secondly, *t' is a suitable subclass of t.* t' must both be the most general subclass of t and the subclassing must be done along the spanning tree introduced by the first parent relation. Where the subclass is abstract, *its* subclass is used, according to the first parent rule above. Thirdly, *t' is in order.* $I(t') > I(k)$, for all $k \in c \backslash t'$. Then, $M' = I(t')$. *c' is a metaknowledge-valid combination.* c' must not contain any abstract classes, must not contain any pair of classes which are from the same mutually-disjoint set, must not contain any pair of classes (r', r'') such that r' is a subclass of r'', and there must be an object in the database of (conjunctive class) t'. P' is set to the first position at which the conjunction c is out of order. $M' = M$. Finally, bounds must be respected; no more than *MaxDC* add or downcasting operations have been performed on the conjunction of which no more than *MaxDCS* have been on simple classes and no more than *MaxDCP* have been on parametric classes. The type specification operator does not change the set of fresh variables.

Taking a simple example, the refinement using class restriction of the the query

$$\mathsf{M[mutagenic \rightarrow yes] \leftarrow M[hasgroup \rightarrow G], M : molecule, G : group} \tag{4.9}$$

includes

$$\mathsf{M[mutagenic \rightarrow yes] \leftarrow M[hasgroup \rightarrow G], M : molecule, G : \underline{ring}} \tag{4.10}$$

Ordered substitution — ρ_{mv} and ρ_{mc}

Where suitable orderings exist between arguments of the literal, refinement may be carried out with respect to the *arguments* in the literal as well as by adding new literals to the clause as a whole. With each such refinement using the arguments of the literal, a refinement operator specialises the clause it operates on, according to some generality ordering. Necessarily, the refined clause covers fewer (or the same number of) examples in the knowledge base. In ordinary refinement, adding a literal establishes a constaint on the clause in order to specify it. An order-yielding method gives rise to a family of such constraints, which take their generality ordering from the ordering relationships between the objects specified as arguments.

Example 4.22 (Ordering of natural number conditions). Consider a method testing the ordering over natural numbers. We might specify an ordered method lessthan with the following signature and implementation:

$$\mathsf{MS = integer[lessthan(integer) \Rightarrow boolean]} \tag{4.11}$$

$$\mathsf{A[lessthan(B) \rightarrow true] \leftarrow A < B.} \tag{4.12}$$

Any clause involving the literal $\mathsf{X : integer[lessthan(2) \rightarrow true]}$ for some preceding variable X of class integer may be refined by replacing the literal with $\mathsf{X : integer[lessthan(1) \rightarrow true]}$ because $1 < 2$ in the related order, and therefore any example covered by the former is clearly covered by the latter. As such, the refinement is a specialisation.

Accordingly, literals produced from order-yielding methods represent conditions which can used to refine features. The method need not be well-ordered; the method may exploit any ordering of objects. These orders are defined by the metaknowledge constructs partialorder and wellorder, introduced in section 3.4.3. From each of these orders, we can compute a set of legal constants to use in order-yielding methods, and use the metaknowledge to refine the constraints they introduce. Under this general scheme, suppose for a constant k, the term used in the next most strict instantiation of the order-yielding method is $succ(k)$. Additionally, the constant used in the most general instantiation of the order-yielding method is k'. Note that an adaptation to the process of literal addition is necessary for order-yielding methods. Namely, in an order-yielding method, the input variable is no longer necessarily one that appears previously in the feature, but a fresh variable. This exception is made to prevent redundancy while still allowing argument specialisation. We consider the general situation of C containing molecules such that: $\mathsf{X[M(Y) \rightarrow true]}$ (or, as frequently-used in practice, $\mathsf{X[M(Y)]}$, where $\mathsf{X : T}$ and $\mathsf{Y : T}$, M is a argument-subsuming method and r is some constant returned from M. We consider two cases. Given a molecule in C acting on an host object of class T with method M and single (possibly ground) input argument A:

Substitution of a variable (ρ_{mv}). If X is a fresh variable occuring in the input argument of an order-yielding method with associated successor function *succ* and most general constant k', obtain D from C by substituting

k' for the variable X appearing in argument A in C.

For example, the refinement of the query

$$\mathsf{M[mutagenic \rightarrow yes] \leftarrow M[hasgroup \rightarrow G], G[atomcount \rightarrow A], A[lessthan(N) \rightarrow true]} \quad \mathsf{M : molecule, G : group, A : integer, N : integer} \quad (4.13)$$

includes

$$\mathsf{M[mutagenic \rightarrow yes] \leftarrow M[hasgroup \rightarrow G], G[atomcount \rightarrow A], A[lessthan(\underline{15}) \rightarrow true]} \quad \mathsf{M : molecule, G : group, A : integer} \quad (4.14)$$

where the maximum number of atoms in a group defined in the corresponding order is 15.

Substitution of a constant (ρ_{mc}). If the feature C was refined in the last refinement step and substituted a term t in an argument A, obtain D from C by substituting t by t' in C, where $t' = succ(t)$ under the order associated with the method.

For example, the refinement of the query

$$\mathsf{M[mutagenic \rightarrow yes] \leftarrow M[hasgroup \rightarrow G], G[atomcount \rightarrow A], A[lessthan(15) \rightarrow true]} \quad \mathsf{M : molecule, G1 : group, A : integer} \quad (4.15)$$

includes

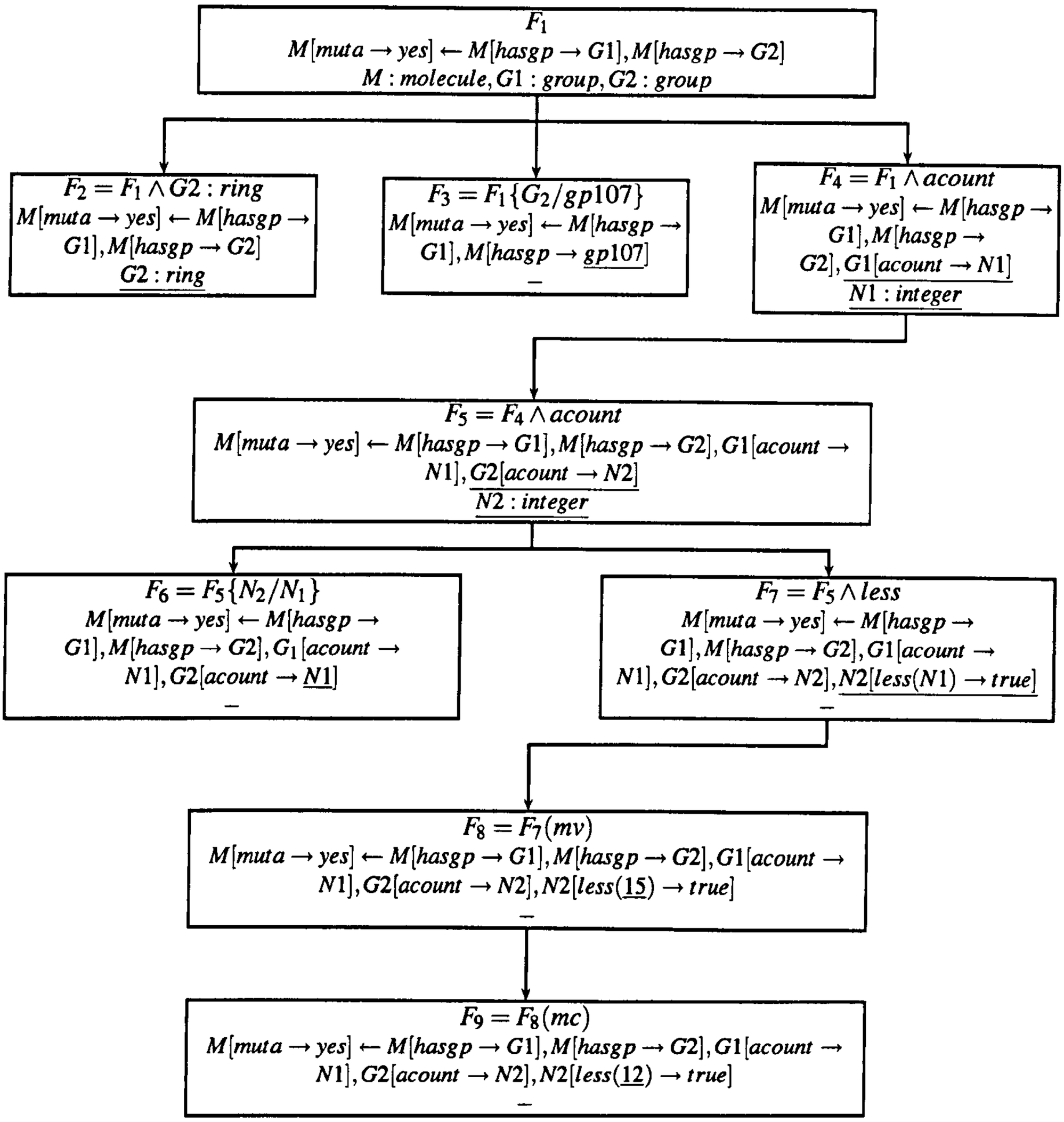
$$\mathsf{M[mutagenic \rightarrow yes] \leftarrow M[hasgroup \rightarrow G], G[atomcount \rightarrow A], A[lessthan(\underline{12}) \rightarrow true]} \quad \mathsf{M : molecule, G1 : group, A : integer} \quad (4.16)$$

where $succ(15) = 12$, which might describe in the context of the metaknowledge that the second-largest number of atoms in a group is 12 (the largest being 15).

Finally, figure 4.3 illustrates a fragment of the refinement graph in the mutagenesis domain which demonstrates the result of applying each constituent refinement operator. Note that the features generated during the refinement are not necessarily complete, *i.e.*, not all variables are consumed.

4.4 Conclusion

This chapter has presented techniques for inductive logic programming, an approach to induction in deductive databases. It began by considering the notion of induction and its relationship to ILP, and presented some settings of ILP which are commonly adopted. ILP relies on the definition of a generality ordering over a space of hypotheses. We reviewed a number of common orderings which are adopted in ILP and how the orderings lead to a lattice structure being induced over a hypothesis space. ILP is typically implemented as a search process, and we considered the general approach of refinement operators for searching the lattice structure and



The background knowledge defined for this lattice is as follows: molecule[muta ⇒ bool], molecule[hasgp ⇒ group], group[acount ⇒ integer], integer[less(integer) ⇒ bool], ring :: group. Clause restrictions are omitted were they do not change between features. group and integer are defined to be subprimitive types. ∧ is an abbreviation for the addition of a literal or class restriction, (mv) for argument specialisation from a variable, and (mc) for argument specialisation from a constant.

Figure 4.3: An fragment of the refinement lattice in the mutagenesis domain showing the behaviour of each refinement operator.

some desirable properties of these refinement operators.

These concepts were then taken into the constrained object logic of CORLOG, and in particular considered the validity of substitutions under the CORLOG framework, identifying variable instantiation, variable unification and class restriction as three types of substitution which are possible. We examined how the result of these substitutions necessarily bring about resulting class constraints when performed under the object model. When performing such substitutions, it is desirable not to produce decomposable features for learning. Accordingly, we examined an approach to substitutions using a graph-based analogy which will not lead to decomposable features. Further restrictions resulting from class-based metaknowledge such as the semantics of parametric classes, total, disjoint and abstract classes were then considered in order to refine in a way which respects this metaknowledge.

The process of search was then considered, and a refinement operator for searching a CORLOG hypothesis space proposed. The refinement operator comprises a number of constituent operators, each fulfilling a seperate aspect of substitution under CORLOG. During search, the refinement operator produces a CORLOG feature set — a set of features which are type-compatible with the target relation and which fulfil some quality criterion. In the learner presented in this thesis, a subset of the features searched by this refinement operator are selected and transformed into a propositional form suitable for an arbitrary attribute-value learner to produce rules, which are in turn then translated back into CORLOG form. Chapters 5 and 6 together situate the use of this refinement operator in a inductive logic programming system.

Chapter 5

Propositionalisation and feature reduction

The previous chapters introduced a representational language for expressing and reasoning with knowledge and queries for object logic and studied approaches to structuring the space of object queries and generating sets of features by means of a special-purpose refinement operator. In order to perform learning with these sets of features, a suitable learning algorithm must be employed. In this thesis we study the application of a transformative approach to such learning, termed *propositionalisation*. A multirelational learner based on propositionalisation, instead of searching the first-order hypothesis space directly, generates a large number of first-order features, uses a transformation module to construct propositional features, and defers the learning task to a propositional learner. We consider propositionalisation as a two-step approach, namely the process of feature construction and the subsequent process of translation into a single-relational problem. Accordingly, we characterise and discuss commonly-adopted aspects of bias which are applied to the language in propositionalisation settings and also formalise the process of transforming a multi-relational problem into a single-relational one. We review closely related systems and introduce the special considerations for propositionalisation in object logic approaches.

While the approach has a number of benefits, the transformation typically results in a dataset of high dimensionality with relatively few examples. An important consideration when mining such data is the obstacle of overfitting the training data as well as the resulting increase in algorithmic complexity. This well-understood problem is known as the curse of dimensionality. Furthermore, features are often highly correlated with each other or redundant in the presence of other features in the training data. The particular acuteness of the curse of dimensionality in propositionalisation means that a feature reduction step is advantageous, both in terms of avoiding overfitting as well as in terms of efficiency.

This chapter considers these two issues together. It formalises the process of propositionalisation, and analyses some novel approaches to post-processing propositionalised data in order to reduce the number of features. In particular, a solution is proposed which aims to detect those features which are logically redundant — those which are useless for discriminating examples of one class from another in the presence of other features — and remove them. Central to this approach is the partitioning of the example set into subsets of one class only, and detecting redundancy between these.

Declaration: Part of this work was carried out in collaboration with Annalisa Appice, Michelangelo Ceci and Peter Flach, and was previously published [5]. The algorithm for neighbourhood decomposition (REFER-N) was presented in this work, with REFER-R having been extended from earlier work by Lavrač, Gamberger and Jovanoski [83]. The remaining work, situating the method, presenting the notion of feature ranking for REFER, the explanation of combined coverage and the analysis of the reason why REFER-N works, is solely that of the author.

5.1 Propositionalisation

Approaches to multirelational learning based on propositionalisation have gained significant new interest in recent years. Propositionalisation is the transformation of a relational learning problem into an attribute-value representation suitable for a conventional, propositional, data mining system. The attributes are also often called features and form the basis for columns in single-table representations of data. Accordingly, propositionalisation is ordinarily applied in domains with a clear notion of individual with learning occurring on the level of individuals only [45].

Where a search of a first-order hypothesis space is too complex or computationally intensive to carry out, propositionalisation systems instead typically employ simpler methods of feature construction, in effect searching a subspace of first-order clauses, defined by the construction method. In this way, the potentially explosive number of possible features involved in a first-order feature search may be contained to a reduced problem which nevertheless retains desirable properties of the first-order equivalent. The resulting features are, in theory, less expressive than their first-order counterparts and may lead to information loss, where the original dataset cannot be recovered from its propositionalised format. De Raedt [28] showed that in the general case, such information loss was inevitable if combinatorial explosion was to be avoided. If we are willing to accept such an approximation for the purposes of learning, however, we gain a number of advantages. As well as computational benefits, a wide range of established propositional learning algorithms may be applied to the transformed data, which are often highly efficient. Moreover, where such learning algorithms produce results in the form of rules, they may be re-expressed as first-order rules by a back-translation. Since such rule learners may use negations of input features, back-translated rules may employ a more flexible form of negation, in which sets of literals in the back-translated rules may appear negated.

In this thesis we consider *query-based transformation* [29], in which a language L represents the set of all possible queries. Such queries are equivalent to the features in the framework presented in this thesis. Transformations may be *complete*, in which the transformation uses the set of all features in L, or *partial*, in which a subset of the features are selected. Partial transformations may also be referred to as *heuristic*, referring to the fact that what is of interest is a small but relevant set of features. Where L is the hypothesis language, a complete tranformation may in theory lead to no information loss, but in practice suffers from the combinatorial explosion discussed earlier. In practice, a partial transformation is performed, bounded by two main user-specified mechanisms. Firstly, the user may engineer L to specify a subset of the hypothesis language which is of interest. Secondly, the user specifies filtering methods based on the quality or interestingness of a feature. Approaches to these techiques classify most current approaches to propositionalisation.

5.1.1 Transformation into attribute-value form

Having informally reviewed approaches to propositionalisation from the literature, we formalise and generalise the notion of the process of propositionalisation and the resulting dataset. Initially we consider the process independent of the choice of logical formalism, later casting the approach to the object logic and its structure described in chapters 2 and 4. Krogel [76] defines propositionalisation as follows:

Definition 5.1 (propositionalisation [76]). Propositionalisation is the process of transforming a relational representation of data and background knowledge into the format of a single table that can serve as the input to a propositional learning system.

We concern ourselves with the ILP predictive learning task in which an intensional definition or hypothesis H is to be learned from examples E, background knowledge B, previously defined in definition 4.3. While this definition described predictive induction in terms of positive and negative examples, we extend the inductive setting such that an example may be tagged with one of a number of class labels. In this reformulation, we attempt to find a theory such that the rules for each class is complete and consistent with respect to the others.

The form of the examples is often restricted from the general ILP setting. Examples in general ILP may take the form of structured facts representing lists and trees, or consist of rules (or facts) which are non-ground. The process of flattening [122, 81] in which clauses involving structured terms are transformed to flattened (functorless) clauses expressed in terms of new functor predicates is often employed prior to propositionalisation. Furthermore, we assume that we are learning with respect to a single-predicate learning task, in which the property of E to be explained by H is in terms of one predicate only.

This predicate is termed the *target predicate* p_t. Each example in E is then represented by an atom of predicate p_t of the form $p_t(X_1,...,X_n,C)$ for an example defined by the arguments X_i and (target) class C. The arguments X_i serve to uniquely identify the example, while C is the target class for classification. Under the object model, a single argument X identifies the example, and so we assume the target predicate to be of the form $p_t(X,C)$, or $\mathrm{X}[\mathrm{p_t} \rightarrow \mathrm{C}]$ in CORLOG. Furthermore, learning problems involving two classes may by represented by the facts $p_t(X)$ (positive examples) and $\neg p_t(X)$ (negative examples).

Each hypothesis $h \in H$ then adopts a *head literal* of the form $p_t(X,Cl\theta)$ for a ground class constant $Cl\theta$ involving a ground substitution θ and variable X. This head literal represents the most general clause in the set of clauses produced by the transformation. In propositionalisation approaches, a process of *feature construction* produces a *feature set* ordered into a feature sequence C, where C is a subset of the language L, with each clause in C having a head of the form $p_t(X,Cl)$. Without loss of generality, each $c \in C$ can therefore be considered a clause body. We are now ready to define propositionalisation in terms of a function. Since the task is a predictive task, each example needs to take into account the class of the example defined by the target predicate p_t. We adapt this framework from [76], specialising it to the setting of the COSINUS learner.

Definition 5.2 (propositionalisation function for an example). Given $C = \langle c_1,...,c_n \rangle$, a feature sequence — generated by a feature construction algorithm, and $v(c,\theta,e,B)$, a general function evaluating feature c for example e against background knowledge B and substitution θ yielding a ground class constant, the *example propositionalisation* P_v of an example $e = p_t(X,Cl)\theta$ is defined as a vector:

$$P_v(C,e,B) = \langle X\theta, v(c_1,\theta,e,B), v(c_2,\theta,e,B),...,v(c_n,\theta,e,B), Cl\theta \rangle \tag{5.1}$$

The substitution θ is such that $e = p_t(X,Cl)\theta$.

Informally, P_v represents a row in the transformed dataset. Since C is ordered, each element of $P_v(C,e,B)$ can be associated with the clause that generated it. The definition extendes naturally to a sequence of examples E, so that $P_v(C,E,B)$ yields a single table in which each row relates to a corresponding element in E, yielding a *propositionalisation* of the examples E.

Definition 5.3 (propositionalisation function for a full dataset). Given C, a feature sequence as above, $P_v(C,e,B)$, a propositionalisation function for examples, and $E = \langle e_1,...,e_m \rangle$, a sequence of examples,

$$P_v(C,E,B) = \langle P_v(C,e_1,B),...,P_v(C,e_m,B) \rangle \tag{5.2}$$

The remaining element of this formalisation is the evaluation function $v(c,\theta,e,B)$ which, informally, determines the value appearing in the table for a given clause c, ground substitution θ, background knowledge B and example e. Typically, the value 1 is used if the example $e = p_t(X,Cl)\theta$ can be proven from the background knowledge B and the clause $c\theta$ substituted with the bindings from the head literal, as in equation 5.3.

$$v(c,\theta,e,B) = \begin{cases} 1 & \text{if } B \wedge c\theta \vdash e \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

Work by Krogel [76] situates evaluation functions[1] in a more general setting, which considers the possible substitutions possible for the variables $Vars(C)$ of a clause C. The background knowledge B is saturated so that it consists of ground facts only and a set $val(C,e,B)$ is defined, consisting of all possible substitutions θ obtained when matching the clause head against the example and the clause body against the background knowledge B. That is,

$$val(C,e,B) = \{(V_1\theta,...,V_n\theta) | C\theta \subseteq B \cup \{e\}\} \tag{5.4}$$

The propositionalisation function is then defined in terms of this set of possible substitutions. Where $|val(C,e,B)| = 0$, no variable matches are possible between C and e with respect to B, and the resulting value is 0. In the existential setting, $|val(C,e,B)| \geq 1$ means that a substitution is possible and a value of 1 is assigned to v. *Counting propositionalisation* records the size of the set, meaning that $|val(C,e,B)|$ is assigned. In the predominant case where v is existential, it is usually more practical to adopt the resolution-based evaluation function described in equation 5.3 and use a logic programming system to obtain the first available substitution, at which resolution is stopped for efficiency. Furthermore, with this approach, background knowledge need not be saturated as in 5.4.

The propositionalisation is now ready to present to an external attribute-value learner. A wide range of possible learners may therefore be applied. Propositional learners may be characterised by the form of the models they produce. *Rule learners* produce rules as their output, typically as conjunctions of literals, where each literal is a feature in the propositionalisation. Other learners, such as neural networks, Bayesian methods, and support vector machines, do not have this property. The rule learners are those of most interest to propositionalisation techniques. The main reason for this is that the literals are expressed as features and therefore the original feature definitions may be substituted in-place to reconstruct a theory in terms of the original model, provided that the variables in the head of the clause match with those in the body. This process is termed *back-translation*, and the result is a self-contained theory which can be used in a logic programming system to

[1] Evaluation functions are termed *propositionalisation functions* in Krogel's work.

evaluate unseen examples.

5.1.2 Feature construction

In propositionalisation, imposing constraints on the hypotheses generated is of particular importance. This section considers the forms of bias employed in feature construction for propositionalisation, and their effects on the language L. Bias in ILP generally was discussed extensively in section 2.2, and so in this section we briefly account for aspects of bias suitable to propositionalisation. We do so with reference to the two mechanisms defining L; firstly, properties imposed on the literals and features in a propositionalisation, and secondly, filtering methods based on the quality or interestingness of a feature. We intentionally adopt a general view of feature construction for propositionalisation in this section.

Individual-centred representations were discussed previously in section 1.2. Their use is central to propositionalisation methods; propositionalisation assumes there is a clear notion of an identifiable individual in the data model so that features can be constructed. In the resulting propositionalised table, each row then relates to an individual and each (non-class) attribute contains the truth value of one of the features for the given individual. Any term in a rule relates to either the individual or an element of it. Structural and property relationships are represented by structural and property predicates. During feature construction, the application of structural predicates always introduces a new variable, and as such is the only way nondeterminacy can occur in an individual-centred representation. Literals based on property predicates consume variables introduced by structural predicates. One stopping criterion for feature construction is when all of a features' variables are consumed in this way. The feature is then termed *complete* or *fully-consumed*.

Adopting the individual-centred representation brings about some of the notions of bias introduced earlier. The provider-consumer model of arguments corresponds closely to the notion of WARMR-style modes in ILP generally, though as Kramer, Lavrač and Flach [72] point out, 'mode declarations constitute a bias for the body rather than a bias for features'. That is, bias is defined on the predicate level rather than the argument level in the individual-centred representation. Related to modes is the requirement that an argument is instantiated, either by the use of a constant symbol in the predicate definitions or by generating patterns tagging arguments as instantiate (e.g. as in RSD). Recall is another important form of bias, corresponding to cardinality constraints. Finally, assigning types to arguments in order to restrict unification of variables and specify a set of substitutable constants for an argument, is common in propositionalisation approaches.

Other language biases have been adopted for defining suitable languages for propositionalisation. Recent work by Železný [142] has considered the application of *feature grammars* in the system EFFEDRIN to define a set of admissible features, combining the stages of feature construction and evaluation in order to arrive at an ordering by which the space may be searched faster. Feature grammars provide a framework for the moding, typing and recall constraints discussed earlier as well as variable, literal and node depth, the latter constraining a depth on the structure of an individual. Orderings on features are defined in order to prune areas of the feature search space, and a refinement of first-order features, by adding literals is arrived at. An admissibility constraint is defined with respect to the grammar, the ordering and bounds on depth and length. Admissible features are also required to respect variable consumption and undecomposability, while being relevant (possessing sufficient coverage) and being non-trivial.

The individual-centred representation adopts a number of important properties of features in propositionalised datasets. Section 3.2 discussed properties of CORLOG features constructed during induction. We briefly

discuss some of these properties in the general context of propositionalisation. Firstly, propositionalisation systems typically restrict features to those which belong to the Datalog class. Arguments to predicates in Datalog adopt only variables and constants as their arguments. Definition 3.6 defined a linked feature. In order to be evaluable and meaningful for propositionalisation, its variables must be linked. The notion of decomposability, defined in definition 3.7, is important. Since the feature acts as an ingredient of a rule, no feature should be expressible as a conjunction of two simpler features already present in the feature set, since they are redundant in the presence of their constituents. In general, ensuring the absence of decomposable features leads to a more compact propositionalisation for the same set of potentially learnable rules. Finally, some learning problems require, or are greatly facilitated by, the ability of the learner to learn recursive clauses — those in which the predicate symbol in the head appears in at least one of the literals in the body. Since this brings about its own complications, recursive features are often assumed to be absent. We made this assumption in section 3.1.

Following from the idea of a linked feature is determinacy, introduced in section 2.2. Determinacy is a common means of limiting a candidate hypothesis space. Determinate features possess only one substitution for their variables, and lead to considerable efficiency during evaluation. In the context of the individual-centred representation, where all the structural properties are determinate, the clause is said to be determinate. Determinacy, ij-determinacy and variable depth form syntactic bounds on clauses in a number of propositionalisation systems. We examine to what extent common forms of bias and these syntactic bounds are adopted in propositionalisation systems in section 5.1.3.

In addition to specifying strict bounds on the permissible features in the language L, heuristics and filters may be applied to generated features to determine whether a feature appears in the feature set. The criteria that these filters adopt can be divided into several categories.

Quality-based heuristics determine a feature's inclusion by some quality measure, often involving the calculation of coverage of (a subset of) the examples. Accordingly, the quality of a feature is often considered according to its correlation with the class. Employing heuristics to test each feature generated is likely to be computationally expensive and re-create problems associated with the explosion in the feature space, although caching can alleviate this problem. *Syntactic filters* determine a feature's inclusion by constraints on the form of the feature or the variables participating in it. These can take the form of simple bounds on the number of variables and literals appearing in a feature, for example. They may exploit metaknowledge about the interactions of variables in methods in order to detect impossible, redundant, or otherwise invalid literals syntactically. An example of this comes from LINUS, in which metaknowledge about the symmetry of predicates reduces the hypothesis space. Alternatively, minimum description length (as in [73]) or other clause complexity bounds could be set. *Logical redundancy filters* determine a feature's inclusion by considering its coverage in the prescence of other features. Necessarily, they are applied after the propositionalisation is complete, as a post-processing measure. Such redundancy filters are discussed in much more depth in the latter part of this chapter, starting section 5.2.

5.1.3 Existing propositionalisation systems

A wide variety of approaches to propositionalisation have developed in the last ten years. Approaches may be general, or specific to a particular aspect of the task. This aspect might be related to the domain, such as computational chemistry [69, 70] Alternatively, it may focus on a particular data structure. In SUBDUE [22], for example, structural concepts relating to graphs, *i.e.* substructures (consisting of subgraphs, not necessarily con-

nected subgraphs), are constructed. The most compressive (with respect to MDL) substructures are compressed in the graph and further iterations build up a hierarchical model of regularities. General-purpose approaches differ most in the method of feature construction, and therefore the choice of language L. A good survey of approaches can be found in [72]. *Pure logic approaches* may operate in a language which is purely first-order, or some other logical formalism (such as F-logic). Features are typically existential, testing whether a feature succeeds for a given example. *Database-oriented approaches* operate more at the database level, taking into account non-existential elements such as database aggregates. *Stochastic approaches* [73] maintain a number of clauses, employing a generational method which removes clauses probabilistically based on their minimum description length while adding new ones constrained by desirable properties. While it lacks completeness in the search, it attempts to find combinations of features which discriminate well. *Wrapper approaches* employ an ILP learner to return a rule, and each subset of literals whose variables connect and which aren't useless (true or false for all examples), are discarded.

In [133], the features from PROGOL are used for multiple linear regression. Similarly, features constructed by the association rule miner WARMR [36] have been used in propositional algorithms [116]. *Lazy propositionalisation* was introduced with the learner PROPAL [39], which deviates from the regular propositionalisation framework by no longer necessarily constructing one tuple per example. An AQ-like algorithm considers the examples most discriminant for the learning task and in so doing aims to overcome the exponential increase in the size of the feature set typical in propositionalisation. A propositionalisation pattern (maximal variablisation of a seed example together with equality constraints from the example) is constructed from the training set and the transformed example is represented by the matchings of the example with the pattern. An example may then be represented by multiple instances of Boolean vectors in the transformed table, representing possible *substitutions*. These represent clauses in a FOL search space, and a resulting partial ordering over the space of positive and negative transformed substitutions allows the extension of overly-general hypotheses. Finally, *bottom-up propositionalisation* [71] limits the language L by taking a data-driven approach. The primary difference is that features — here fragments of a molecular structure — are constructed from examples. An extraction algorithm combines with an efficient evaluation algorithm, which, using a frequency-based pruning mechanism is able to construct and evaluate complex, expressive features. Of these, the approaches based on logic are of most interest to the problem of data mining for object-oriented models. Object representations may also be considered as databases defined by strict schemas and so database methods are also of interest. In this section we review methods from these two categories.

The pioneering propositionalisation system was LINUS [77, 80], introduced in 1993. The original LINUS system had little support for the generation of features as they are discussed here. Transformation was performed by considering only possible applications of background predicates on the arguments of the target relation, taking into account the types of arguments. Accordingly, the clauses in L that it could learn were *constrained*; all the variables in the body of the clause also necessarily appeared in the head. DINUS [77] ('determinate LINUS'), a later system, relaxed the bias so that non-constrained clauses could be contructed, given that the clauses involved were determinate; that the variables involved in each literal has only one binding given the binding of the literals that appear before it. While few real-world domains are easily represented by solely determinate predicates, DINUS' ability to learn recursive rules presented a great step forward. SINUS [74], a predecessor to COSINUS[2], was first implemented by the author as an intended extension to the original LINUS as a means of incorporating logic programming feature generation mechanisms for structured

[2]COSINUS stands for 'constrained object SINUS.'

```
--INDIVIDUAL
train 1 train cwa
--STRUCTURAL
train2car 2 1:train *:#car * cwa
car2load 2 1:car 1:#load * cwa
--PROPERTIES
cshape 2 car #shape * cwa
clength 2 car #length * cwa
...
lnumber 2 load #numberl * cwa
```

Figure 5.1: A fragment of the PRD file describing the trains example

domains in a modular system integrated with the propositional learner. Whereas LINUS directly applies background predicates based on the arguments of the target relation, SINUS employs flattened, individual-centred data representations in feature construction, recursively building a clause from structural and property predicates. SINUS thus escapes LINUS' behaviour of constructing constrained clauses only, which follows from its bias. The *extended* LINUS approach [81] is another example of an extension to LINUS which uses structural and property predicates. The main differences lie in the treatment of property predicates[3], the ability to defined biases based on reuse of variables and the choice of stopping criteria[4]

SINUS associates types with the arguments of predicates via a predicates description file, recursively generating first-order features based on the linkage of arguments of the same type and constants appearing in the arguments in the background data. The feature set is used as the basis for propositionalisation, the learner invoked, and the induced model translated back into Prolog form. SINUS is an example of a learner adopting the individual-centred representation, in which an individual is represented by a type, and data is described with flattened Prolog clauses describing structural and property relationships. Data definition is by means of an adapted PRD file (as in the first-order Bayesian classifier 1BC [46]), which gives information about each predicate used in the facts and background information. Each predicate is listed in separate sections describing individual, structural and property predicates. This is used both to define the hypothesis language and the constants associated with each type, which are extracted automatically from the data but may be overridden. A PRD file defining the domain for the trains example is shown in figure 5.1. Each predicate is defined by its name and number of arguments. The types of each argument are given together with their modes and, in the case of structural predicates, their cardinality. The syntax `* cwa` appears to retain compatibility with PRD files.

SINUS constructs features left-to-right, starting with a single literal describing the individual (for example $train(X)$). The construction works recursively, at each stage adding a literal by applying a predicate to the unbound variables in the partially-constructed clause such that the arguments match the types of currently unbound variables. An argument-matching algorithm considers each argument in turn considering the types of the arguments and the current partial binding of arguments to variables, discovering all possible bindings. The bounds *MaxL* (the maximum number of literals), *MaxV* (the maximum number of variables) and *MaxT* (the maximum number of values a type appearing in a property predicate may take) apply to the feature construction. During construction, where an argument is an input, an existing variable is substituted for it. Depending on the model, output arguments are substituted with a new variable or a constant — once for each value the type of the constant may take. Recursion ends when there are no longer any unbound variables. However, this partial

[3] In extended LINUS, arguments may either be a previously unbound variable or a constant of the argument type

[4] In LINUS the number of variables and utility predicates are limited whereas in SINUS the bound is the number of literals and variables.

clause can go on to form new clauses if the user has decided to *reuse* consumed variables. This method of feature generation defines the language L.

The last decade has seen a large number of additional propositionalisation systems proposed, many of which assume an individual-centred representation and the notion of a first-order feature. *Logic-oriented techniques*, such as SINUS and the subgroup discovery system RSD [84] typically produce existential logic programming clauses as part of their central feature construction step. RSD was originally intended as a system for relational subgroup discovery, though we consider it here as a a first-order feature constructor. Like SINUS, features are exhaustively generated via a recursively-implemented search of the hypothesis space of first-order features generated within user-defined bounds. RSD first constructs complete features, and then permutes the constructions with possible variable bindings before generating the data. Constraints are introduced in order to determine which variables are instantiated. Logically redundant features may be detected in order to reduce the feature set size without loss of information or learnability of any possible theory. RSD does this implicitly by testing for equivalency of features instead of post-processing with a feature reduction step. Predicate definitions are used in order to guide and constrain the recursive process of feature generation. As in SINUS, strong typing and input/output-style modes are defined for arguments, and in addition, RSD uses a recall parameter to specify the number of successful calls to a predicate, as in ALEPH [132]. RSD allows user-defined bounds on the depth of variables and number of occurences of a symbol.

Database-oriented techniques, such as RELAGGS [75], operate more at the database level, adding non-existential features such as aggregation when constructing features. In the database-oriented propositionalisation techniques, features are based on the application of database aggregation functions, an approach which is not necessary based on the testing the existence of some property (or properties) of an object associated with the individual under test. An example of a database-oriented system is RELAGGS [75], in which the declarative bias is based on the definition of a database schema, in particular the foreign-key relationships and the indexing used for records. Parallels between such aggregation functions can be drawn with the object view. Aggregation is applied to single fields and pairs of fields in the database and record identifiers are propagated across relational tables by foreign-key relationships. The application of these aggregation functions, for example counts of objects satisfying some condition, has shown more appropriate in some domains [74] than an equivalent purely logical approach.

5.1.4 Propositionalisation in the object model

The propositionalisation framework presented above is largely independent of the choice of logical presentation; the definitions and arguments are well-accommodated by an object logic such as CORLOG. We restrict our attention to features with values 0 or 1 in the propositionalised dataset. We adopt the evaluation function in equation 5.3. That is, given a set of features C sharing a head variable representing the individual X, we adopt the value 1, associating it with the individual given by X, if $B \wedge c\theta \vdash e$ for some feature $c \in C$ and example e such that $e = p_t(X,Cl)\theta$, and 0 otherwise. That is, the feature is true if the body unifies with the example described by the individual identifier $X\theta$ under some substitution θ' based on θ. Otherwise, it is false.

Having assumed that all features are Boolean in this way, we refine the definition of a CORLOG feature to a Boolean CORLOG feature which follows directly from the definition of a CORLOG feature in 3.2. A Boolean CORLOG feature simple restricts the interface to one in which there are no input arguments other than the host argument and no output arguments.

Definition 5.4 (Boolean CORLOG feature). A *Boolean* CORLOG *feature* is a CORLOG clause of the form

$$O[f_a] \leftarrow l_1 \wedge \ldots \wedge l_n \wedge c_1 \wedge \ldots \wedge c_m \tag{5.5}$$

where O is an id-term representing the individual. The meaning of O, f_a, l_i and c_j is as in definition 3.1.

O, as a term in the feature, is itself constrained to a class by a class membership assertion in the conjunction $c_1 \wedge \ldots \wedge c_m$. The notion of a feature set in 3.3 follows from this class of features. Since there is no input or output arguments, the only criterion for a set of features to form a feature set under a common interface is in terms of O. A feature set FS is associated with an interface $int_{FS} = \langle O_{FS} \rangle$. For each feature in this set, the term O is constrained to be of class O_{FS} or a subclass of O_{FS}. A feature is therefore a nullary method on a single individual represented by an id-term. We may occasionally commit a slight abuse of notation and use the term f_a to refer to the body of the feature — $l_1 \wedge \ldots \wedge l_n \wedge c_1 \wedge \ldots \wedge c_m$ — especially in the context of a feature set under a constraint O_{FS}. Furthermore, a valid Boolean CORLOG feature is defined by the same properties as a standard CORLOG feature. More specifically, the feature is linked as in definition 3.6, undecomposable as in definition 3.7 and class-constrained as in definition 3.12.

The remaining issues in carrying propositionalisation over to the object model exist as a result of the two mechanisms identified in feature construction — namely, restricting the language and choosing bias, and it is in these areas that the majority of this thesis is based. We conclude this discussion of propositionalisation by making some further observations. First and foremost, the object model may be seen as a (very much larger) superset of the PRD style of domain modelling for the individual-centred representation, and example of which is shown in figure 5.1. While the PRD representation expresses individuals (objects), part-whole relationships, properties, and cardinalities, and input/output and mode constraints, the object model adds inheritance, finely-tuned value constraints, method meta-knowledge, and more. Secondly, ordinary propositionalisation introduces a simplistic notion of type, which serves to specify unification constraints and the set of values for which an variable argument may be instantiated with constants. Object logics employ a much more sophisticated, and usually finely-grained, notion of type. Mode declarations, on the other hand, have a simple mapping to a method call, which presupposes the host object and arguments are instantiated. Finally, the declarations serve as a convenient *restriction* of the language L. Accordingly, object declarations should also seek to control the nature of the search through the hypothesis space.

5.2 Feature elimination for propositionalisation

Under the setting presented, the resulting dataset belongs to a specific class of dataset, which we refer to as a *propositionalised dataset*.

Definition 5.5 (propositionalised dataset, feature, class). A *propositionalised dataset* is a collection of n training examples $e_i \in E$ which are described by m Boolean features $f_i \in F$ where a *feature* is a mapping from examples to Booleans. Furthermore, each example is labelled with a *class*, drawn from a set C of possible classes.

For the rest of this chapter, the term *dataset* will be restricted to only propositionalised datasets. In the case where $|C| = 2$, we term the dataset a *two-class dataset*. Where $|C| > 2$, the dataset is termed a *multi-class dataset*. Furthermore, we adopt predictive ILP as the normal setting for propositionalisation, defined in

definition 4.3 and extended to the multiclass setting in section 5.1.1. Accordingly, we assume that the features in the dataset being reduced are to be used in classification rules. The hypothesis language being searched is therefore assumed to be the set of all possible conjunctions of features.

5.2.1 Detecting redundancy in example set partitions

Informally, a feature is said to be logically redundant if there exists another feature in the data which discriminates at least the same examples of one class from another for the purposes of classification rule learning. More formally, the definition of logical redundancy comes from the work on the REDUCE algorithm [83], adapted here to the multi-class case by considering redundancy between classes.

Definition 5.6 (logical redundancy, logical redundancy for multi-class settings). A feature f is *redundant* in the presence of another feature g if g is true for at least the same positive examples as f and false for at least the same negative examples as f. A feature f to be *redundant for discriminating class a against class b* in the presence of another feature g if g is true for at least the same examples of class a as f and false for at least the same examples of class b as f. Where a feature f is said to be redundant in the presence of g, we say that g *covers* f.

We refine this concept further to give a formal definition of coverage, following that given by Lavrač *et al.* [83] for the two-class case:

Definition 5.7 (coverage). Let E_l and E_m be a pair of subsets of E such that every example in E_l is of class c_l and every example in E_m is of class c_m ($c_l \neq c_m$). A feature $f \in F$ *covers* a feature $g \in F$ *for discriminating class c_l from class c_m* for the examples $E_l \cup E_m$ if $T_l(g) \subseteq T_l(f)$ and $F_m(g) \subseteq F_m(f)$, where $T_l(f)$ is the set of all examples $e_i \in E_l$ such that f has the value *true* for e_i and $F_m(f)$ is the set of all examples $e_j \in E_m$ such that f has the value *false* for e_j. Similar definitions follow for $T_l(g)$ and $F_m(g)$.

Informally, for classification rule learning, we assume that feature f is better than another feature g for distinguishing c_l from c_m if f is true for at least the same c_l examples as g, and false for at least the same c_m examples as g. In this setting, we nominate class c_l as the class which the classification rule attempts to describe. The notion of a useless feature follows from this definition.

Definition 5.8 (useless feature). A feature f is said to be a *useless feature* for a subset E_i if either $T_i(f) = \emptyset$ or $F_i(f) = \emptyset$.

Returning to the classification rule basis for feature reduction, since the conjunctions of features in the hypothesis language L_H are not necessarily assumed to include negations of features, it is therefore necessary for the user to provide additional negated features where this is required. In some learners, negation may be eventually introduced by the learner in the case where features represent only positive literals. Then, there is an effective comparison between f and g as well as $\neg f$ and $\neg g$, though not between f and $\neg g$. If negated features are supplied explicitly, all possible combinations are evaluated.

Useless features can be immediately removed from the set of features F regardless of the values of other features, since they do not discriminate c_l from c_m for any examples. They are therefore of no use in a classification rule and can be eliminated in preprocessing without compromising the discovery of a complete and consistent hypothesis from the reduced data. A feature being detected useless can be considered a type of

Algorithm 5.1: REFER-R: feature elimination algorithm.

input : E_l, a set of examples of class c_l
input : E_m, a set of examples of class c_m
input : F, a set of features
input : R, a set of features from F already detected as non-redundant
output: RF, the set of features detected non-redundant for discriminating class c_l against class c_m

$RF \leftarrow F \backslash R$
forall $f_i \in RF$ **do**
  if f_i *has value false for all examples* $e \in E_l$ **then**
    remove f_i from RF
  if f_i *has value true for all examples* $e \in E_m$ **then**
    remove f_i from RF
  if f_i *is covered by any* $f_j \in RF$ **then**
    remove f_i from RF
forall $f_i \in RF$ **do**
  if f_i *is covered by any* $f_j \in R$ **then**
    remove f_i from RF
return RF

logical redundancy. Where a feature is detected as useless, it can be considered redundant without needing to check other features for coverage.

Given an arbitrary dataset, we therefore detect the redundant features by first checking for uselessness and then checking pairwise against other (remaining) features for coverage. This algorithm, REFER-R, is expressed in pseudocode in algorithm 5.1. Note that it allows for the input of a a set of features to be marked as already non-redundant.

REFER-R is similar to the REDUCE algorithm, but extends its setting to that of datasets taking more than two classes. The notions of positive and negative examples in REDUCE correspond to examples of class c_l and c_m, which may be any two possible classes. Further, the algorithm allows a number of features already found non-redundant to be supplied. The algorithm does not consider these for removal, but instead first collects all candidate features — those not yet marked as non-redundant. Like in REDUCE, if a feature in RF is covered, or it is true in its c_l examples and falso in its c_m examples, it is removed. However, following this, it checks each feature against those that have already been marked non-redundant (in R). The transitivity of the coverage relation ensures that this leads to no drop in performance, and furthermore, requires fewer comparisons.

This approach is of use in the multi-class case, in which REFER-R is used to find non-redundant features between each pair of classes (l,m) with corresponding sets of examples (E_l, E_m). A feature is then determined non-redundant if it has been found non-redundant for (at least) one pair. Necessarily, REFER-R is applied multiple times, and R accumulates the non-redundant features from each application.

5.2.2 Partitioning and comparing pairs of partitions

Fundamental to the feature reduction approaches discussed in this chapter is the set partitioning of the dataset into *neighbourhoods*, where each example in a neighbourhood shares the same class label. These are often sets of examples where the number of features differing in value between the examples is relatively small. This partitioning has two important effects. Firstly, it allows the reduction of multi-class data, and secondly, by

Algorithm 5.2: REFER: top-level algorithm

input : E', a set of training examples
input : F, a set of features
output: RF, the set of reduced features

begin
  $RF \leftarrow \emptyset$
  $E = \{E_1, ..., E_w\} \leftarrow$ PARTITION (E')
  forall $E_i \in E$, *where* E_i *belongs to class* c_i **do**
    forall $E_j \in E$, *such that* E_j *belongs to class* c_j, $c_j \neq c_i$ **do**
      $RF \leftarrow RF \cup$ REFER-R (E_i, E_j, F, RF)
  return RF
end

partitioning we obtain a much-reduced dataset. We define an example set partition as follows:

Definition 5.9 (example set partition, neighbourhood). An *example set partition* E of an example set E' is a set of non-empty disjoint subsets $\{E_1, E_2, ..., E_w\}$ $(w \leq n)$ whose union is E and for each E_i, all the examples are of the same class. Each subset E_i is termed a *neighbourhood*.

From now until the end of this chapter, the term *partition* will be used to describe an example set partition. Similarly to the case in which class-pairs are compared, after an example set partition is constructed, in order to remove redundant features from the example set as a whole, it is sufficient to detect redundancy among each pair of subsets (E_i, E_j) of differing class. Where a feature is found to be non-redundant between any pair, it can be considered non-redundant for the whole example set E'. The set of features found redundant is dependent on the partition, however, and may not be equivalent to using the strategy of comparing, for each class pair, *all* examples belonging to each. By considering each pair of subsets of differing class from the partition, and applying the REFER-R algorithm to detect non-redundant features between them, adding each set to the set for the entire dataset, we arrive at a algorithm 5.2. We represent an arbitrary partitioning algorithm by the symbol PARTITION, deferring discussion of the partitioning strategy until later in this chapter. PARTITION returns a set of example sets, each of which necessarily belong to one class.

Furthermore, it can be shown that for an arbitrary set partition, the removal of redundant features in this way conserves the learner's ability to discover a complete and consistent hypothesis in the reduced dataset, if that were possible in the original data. More formally,

Theorem 5.10. *Given L_H, a hypothesis language rich enough to allow for a theory T, that is complete and consistent for each class, to be learned from a training set E', a set of features F and $\{E_1, ..., E_w\}$ a partition of E', a complete and consistent theory T can be found using only features from the set $F' \subseteq F$ if and only if for each possible pair of examples $(e_i, e_j) \in (E_l, E_m)$ with $c_l \neq c_m$, there exists at least one feature $f' \in F'$ such that $e_i \in T_l(f')$ and $e_j \in F_m(f')$.*

Proof. Necessity: Suppose that a pair of examples $(e_i, e_j) \in (E_l, E_m)$ with $c_l \neq c_m$ exists such that there is no feature $f' \in F$ for which $e_i \in T_l(f')$ and $e_j \in F_m(f')$. This implies that no rule involving features in F' could discriminate between e_i and e_j and a description which is complete and consistent with respect to the class c_i cannot be found. *Sufficiency:* Let $G = (N, A)$ be the graph corresponding to the partition of E', in which edges exist between neighbourhoods of differing class. Consider an arbitrary example $e \in E'$. We can build

a description of e from all those features F_e appearing on all arcs of G connecting its containing node, since we have associated with these arcs the set of all features discriminating e against every other example of a different class, and vice-versa. This description is exactly the conjunction of all those features which are true for e with the negation of each of those features that are false for e. This description is then true for e and no other example. Now consider all those examples in the neighbourhood (and therefore of e's class). We build a description for those examples by constructing a disjunction of the descriptions for each example. This description is true for each of these example and no other example of a different class. Similarly, we can build a description of each class as a whole by taking a disjunction of such descriptions for each neighbourhood. We now have a complete and consistent hypothesis for each class. □

5.2.3 Ranking features for tie-breaking

Where a dataset is partitioned into many neighbourhoods, often containing few examples per neighbourhood, it is possible that within many of the neighbourhood-pair comparisons there exist a large number of features covering all others. These features are then necessarily identical (and therefore cover each other). Where none of these features have previously been found non-redundant, this raises the question of which one of these features to select and determine non-redundant in a particular neighbourhood-pair comparison. Our aim to is achieve the smallest reduced feature set. Over many such neighbourhood-pair comparisons, the feature to select becomes an important consideration. Following the original algorithm, features are ordered by their appearance in the datasets, *i.e.*, feature f_1 may be chosen over an equally-covering feature f_2.

By precomputing coverage relationships between features in each neighbourhood pair, it is possible to rank the features according to the degree to which they are covered by other features, before the execution of REFER-R.

Definition 5.11 (feature ranking score, feature ranking). The *feature coverage score*, denoted $fcs_E(f)$ of a feature f relative to an example set partition E is the sum of $|E_i||E_j|$ for all (i,j) such that f is covered by some g in (E_i, E_j)[5]. A *feature ranking* $R(E)$ for an example set partition E is a sequence of features such that where a feature f exists earlier in the sequence than a feature g, $fcs_E(f) \geq fcs_E(g)$.

Informally, we assume that the higher the score $fcs_E(f)$, the more likely the feature f will be covered in neighbourhood-pair comparisons. By adopting this ranking, in the case of a tie, we aim to remove the feature which is more likely to be also removed in other neighbourhood-pair comparisons. This approach is more likely to yield a greater reduction.

The score is a weighted sum of the number of times a feature is covered by at least one other in each neighbourhood-pair of differing class. The sum is weighted by the product of the sizes of the neighbourhoods. This makes the magnitude of the measurement independent of the choice of partitioning and in particular the size and number of neighbourhoods. Additionally, the measurement is more influenced by larger neighbourhoods, where coverage between features is less common, but where these coverage relationships are more significant in terms of the entire dataset and its reduction. In this way, the coverage relationships in the larger neighbourhoods inform the selection of features between smaller (or singleton) neighbourhoods.

The pseudocode for the algorithm is shown in algorithm 5.3. The RANKFEATURES algorithm closely resembles that of the REFER algorithm, since, with the exception of the availability of the features already

[5] If f is useless, it is considered to be covered.

Algorithm 5.3: RANKFEATURES: Pseudocode for feature ranking

```
input  : E, a set partition of training examples into neighbourhoods E_i
input  : F, a set of m features
output: R, a ranked sequence of features from F

forall 1 ≤ i ≤ m do fcs_E(f_i) ← 0
forall E_i ∈ E where E_i belongs to class c_i do
    forall E_j ∈ E such that E_j belongs to class c_j ≠ c_i do
        forall f ∈ F do
            if f false for all E_i or true for all E_j then
                fcs_E(f_i) ← fcs_E(f_i) + |E_i||E_j|
            else
                if ∃g such that f is covered by g then fcs_E(f_i) ← fcs_E(f_i) + |E_i||E_j|
R ← F ranked according to decreasing fcs_E
return R
```

found non-redundant, they perform the same test. As a result, in the worst case, ranking the features will cause only a doubling in the execution time for any size of example set or feature set. Unfortunately, it is not sufficient to record the fact that a feature is covered in RANKFEATURES and subsequently use all of these facts in the REFER-R run, since in the case where exactly two features are equal, both would be determined covered. In REFER-R, when one is deleted, the other is no longer covered. However, efficiency gains are likely to be possible if coverage of non-equal features was cached, owing to the transitivity of the feature coverage relation. If a feature is covered by a non-equal feature, it will necessarily be removed by REFER-R.

The algorithm produces a ranking of features R, with the first feature in the list being the one covered most often. When considering which of a pair of features to eliminate (and determine redundant), REFER-R will favour the first feature in the ranking, corresponding to the feature which is most covered by the others overall. In practice, the REFER-R algorithm considers features for removal according to the sequence R, thereby eliminating the most covered features first. In chapter 7, we will empirically determine the utility of the REFER algorithm with and without ranking enabled.

Having considered the process of removing redundancy given a particular partition, we next consider the method used to generate the partition — the process of neighbourhood construction.

5.3 Neighbourhood construction

Partitioning an example set is clearly a complicated challenge. Although the problem concerns only the partitioning of the examples of each class, an exhaustive search of the possible partitions is made computationally prohibitive as a result of the combinatoric explosion of the number of possible partitions, given by the nth Bell number B_n [121], where $B_0 = B_1 = 1$ and $B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$. We may draw a parallel between the problem of partitioning and *clustering*, a technique which aims to partition a set of examples into subsets so that the examples in each subset are similar, for some, possibly specialised, notion of similarity. However, effective partitioning seeks instead to combine examples which work together to cause feature coverage. To motivate our choice of partitioning method, we first determine the nature of the conditions to be met by a desirable partition. We then consider a partitioning method — REFER-N— and analyse it according to these conditions.

E_b	E_a		
	subset condition	*constant false*	*constant true*
s.c.	$\exists g, g'$ st $T_a(g') \subseteq T_a(f) \subseteq T_a(g)$, $T_b(g) \subseteq T_b(f) \subseteq T_b(g')$ (subset/subset)	$\exists g$ st $T_a(f) \subseteq T_a(g)$, $T_b(f) = \emptyset$, $T_b(g) = \emptyset$ (subset/constant)	$\exists g$ st $T_b(g) \subseteq T_b(f)$, $F_a(f) = \emptyset$, $F_a(g) = \emptyset$ (subset/constant)
c.f.	$\exists g$ st $T_a(f) = \emptyset$, $T_b(f) = \emptyset$, $T_b(f) \subseteq T_b(g)$ (subset/constant)	$T_a(f) = \emptyset$ and $T_b(f) = \emptyset$ (false feature)	$T_a(f) = \emptyset$ and $F_a(f) = \emptyset$ (impossible)
c.t.	$\exists g$ st $T_a(g) \subseteq T_a(f)$, $F_b(g) = \emptyset$, $F_b(f) = \emptyset$ (subset/constant)	$F_b(f) = \emptyset$, $T_b(f) = \emptyset$ (impossible)	$F_b(f) = \emptyset$, $F_a(f) = \emptyset$ (true feature)

Table 5.1: The result of combining and simplifying conditions for combined coverage

5.3.1 Conditions for combined coverage

We claim in this chapter that partitioning datasets leads to a greater reduction of the feature set. Previous work has shown empirically [5] that the adoption of REFER-N for partitioning leads to a greater reduction than without partitioning. We now analyse the reasons for this increased reduction. In order to do so, we first consider the conditions under which a general partition leads to greater reduction.

Recall that a feature is non-redundant if it is found to be non-redundant (not covered) in at least one neighbourhood-pair comparison. Therefore, a feature f is only redundant if it has been found to be covered in each neighbourhood-pair comparison. By partitioning the training examples into neighbourhoods, f may be covered by *different* features in each neighbourhood-pair comparison, reducing the feature set further. Were the training examples not partitioned into neighbourhoods, it would be necessary for a single feature g to cover f all examples, a condition that is satisfied far less frequently. We call this effect *combined coverage*. More formally,

Definition 5.12 (covered by combined coverage). Given a example set partition $E = \{E_1, ..., E_w\}$, resulting in a set of neighbourhood-pair comparisons $NP = \{(E_i, E_j) | c_i \neq c_j\}$, a feature f is *covered by combined coverage* if f is covered by at least one feature g_i for each compared neighbourhood-pair $(E_i, E_j) \in NP$.

When coverage is being detected over examples in a neighbourhood-pair consisting of examples of class c_l and c_m ($c_l \neq c_m$), coverage is first detected for discriminating c_l from c_m, and then vice-versa. In order for a feature to cover another, conditions must be met for both the c_l and c_m examples in the neighbourhood-pair. Therefore, in order to maximise the effect of combined coverage, it is necessary to consider the coverage relationship in both directions simultaneously, rather than testing them independently.

We summarise the conditions for redundancy as follows, in which $T_a(f)$ is the set of all examples in E_a such that f has the value true for f, and $F_b(f)$ is the set of all examples in E_b such that f has the value false for f. Redundancy is discussed in terms of subset relations, and to simplify discussion, we often adopt the relation in terms of T-sets. Observe that $F_i(f) \subseteq F_i(g)$ iff $T_i(f) \supseteq T_i(g)$ for any E_i.

Recall that a feature f is redundant (with respect to another feature g), where (subset condition) $T_a(f) \subseteq T_a(g)$ and $F_b(f) \subseteq F_b(g)$, or equivalently, $T_a(f) \subseteq T_a(g)$ and $T_b(f) \supseteq T_b(g)$; (constant false) $T_a(f) = \emptyset$; (constant true) $F_b(f) = \emptyset$. By considering each possible combination of these conditions, applied not only for discriminating the examples of E_a (class c_l) from those of E_b but also for those of E_b from those of E_a, we arrive at conditions for combined coverage in both directions simultaneously. Table 5.1 shows the result of combining and simplifying these conditions, incorporating the rule rewriting subset relationships in terms of the T-values.

Ignoring the impossible conditions, the table shows three general forms of these conditions between a pair of features. Firstly, *constant* features are always true or false. *Subset/constant* features, are those which have a constant coverage relation on one side and a subset coverage relation (with respect to some feature g) on the other side. *Subset/subset* features are those with a subset coverage relation (with respect to two features g, g'). With the exception of the last condition, these conditions are computed easily, since each of these conditions are constructed from tests of coverage contained within one neighbourhood. Accordingly, the subset relationships in the second set of conditions need only ever be tested once per neighbourhood and the coverage computed conditional on whether the features on the other side are constant. These subset relationships and constant properties may easily be stored in a lookup table. Similarly, the first set of features may be tested by checking these tables.

In summary, partitioning the example set aids reduction because it relaxes the condition by which a feature can be considered covered. Specifically, it allows a feature to be covered by a different feature in each neighbourhood-pair. Additionally, we saw that most of the conditions for which combined coverage applies depend only on coverage within a single neighbourhood.

5.3.2 REFER-N: Partitioning based on Hamming distance

Having discussed the detection of logical redundancy within a single neighbourhood and the interplay of neighbourhoods comprising a set partition, we discuss methods of arriving at such a partition. Although partitioning the dataset and reducing it by analysing its neighbourhood-pairs means that more features can be found redundant, not every example set partition leads to the same reduced feature set size. In order to determine how best to partition a dataset, the three following desirable properties of neighbourhoods are established.

Firstly, there should be as many coverage relationships as possible between feature-pairs in individual neighbourhoods. Secondly, there should be as many features as possible whose values are constant across the examples in the neighbourhood. However, according to the conditions derived above, if these properties worked together to make a feature redundant, they would need to apply to the same feature on both sides, and so ideally it is necessary to ensure these conditions hold on both sides by explicitly testing them. Thirdly, neighbourhoods should not be too small. The neighbourhood partition for which the most coverage relationships exist is that in which each example has its own neighbourhood, termed *singleton neighbourhoods*. In principle, using this example set partition, it is possible to maximally reduce the feature set. In this extreme case, for each neighbourhood pair, the only non-useless feature is one which is true for the positive class and false for the negative class, according to the definition of coverage. The algorithm then has the task of determining, in $O(|E|^2)$ neighbourhood-pair comparisons for an example set E, which of these many features is to be labelled non-redundant. The introduction of the alternatives leads to the reformulation of the problem at that point to the NP-hard calculation of the minimum hitting set, the minimum reduced feature-set such this reduced set contains at least one element from each neighbourhood-pair comparison. In other words, further decomposition of a (non-singleton) neighbourhood in which there is a clear coverage relationship between two given features, may lead to more features being labelled non-redudant as a result of their consideration separately, and a lesser reduction in the number of features would result. Adopting ranking methods may assist this, however. As well as producing more non-redundant features, adopting singleton neighbourhoods produces many more neighbourhood-pair comparisons and therefore introduces efficiency concerns.

The choice of neighbourhood decomposition is therefore subject to a tradeoff between minimality and the

Algorithm 5.4: REFER-N: neighbourhood construction algorithm.

input : E', a set of examples
output: $E = \{E_1, ..., E_w\}$, a set partion of neighbourhoods in E'. Each E_i is a set of examples tagged with a class c_i.

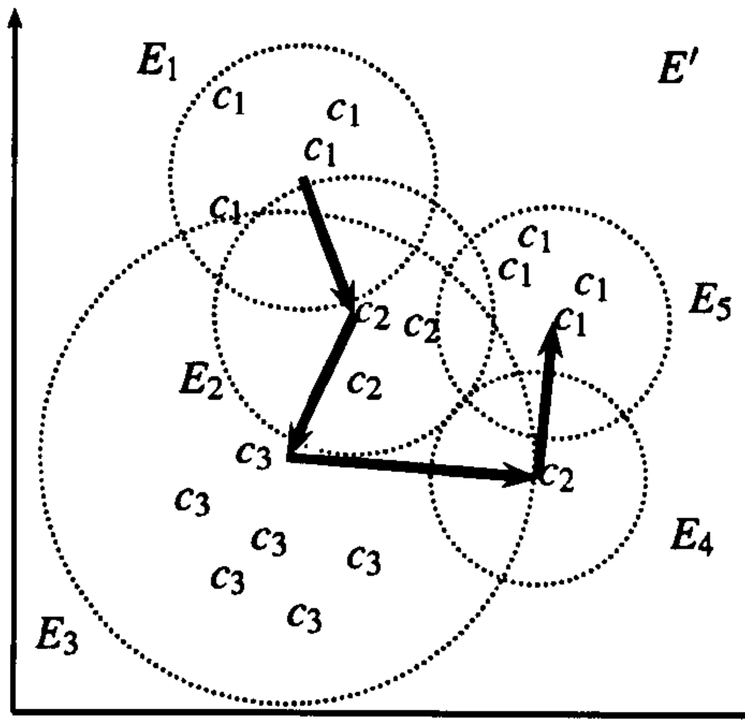
$R \leftarrow E'$; $i \leftarrow 1$; $e_s \leftarrow$ randomly-selected starting example in E'
while $|R| > 0$ **do**
  if *there exist examples not of the class of* e_s *in* R **then**
    $e_t \leftarrow$ closest example in R of a different class according to Hamming distance
    $E_i \leftarrow E(e_s, e_t)$ where $E(e_s, e_t) = \{e' \in R | d(e_s, e') \leq d(e_s, e_t)\}$ where d is the Hamming distance
    $R \leftarrow R \backslash E_i$
    $e_s = e_t$
  else
    $E_i \leftarrow R$
  if $|R| > 0$ **then**
    increment i
  $c_i \leftarrow$ class of e_s
return $E = \{E_1, ..., E_i\}$

complexity of the final selection task of finding the minimal set from the features found in each neighbourhood pair, although the adoption of the ranking technique for features can help alleviate this

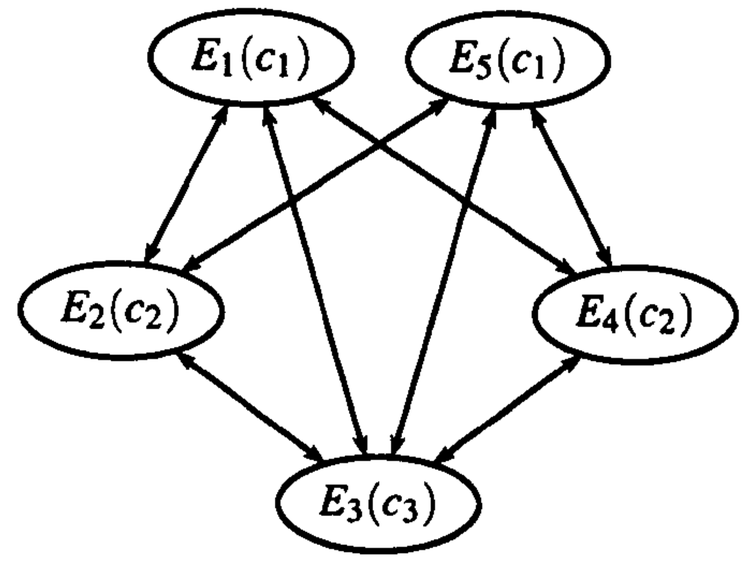
Clustering the examples by the number of features in which they differ — the Hamming distance — is a simple but effective approach to example set partitioning. By producing neighbourhoods in which examples are within a limited Hamming distance of each other, we encourage the incidence of features which have the same value for every example in the neighbourhood, although there is less control over the particular feature positions at which these differences occur. We demonstrate that a particular approach to this Hamming-distance-based partitioning promotes coverage among the features.

In REFER-N, the examples are considered as points in an n-dimensional Hamming space, or the set of all binary strings of length n. The Hamming distance, defined as the number of positions for which the strings differ, is used as a metric on this space. Each neighbourhood can be uniquely identified by two examples, the first being the centre of a neighbourhood in the Hamming space, chosen as a starting point for the construction of a neighbourhood. The second is an example of another class, acting as a termination point for the neighbourhood's construction.

The REFER-N method is shown in algorithm 5.4. In it, we select a random example $e_s \in E'$ as a starting example for the construction of a neighbourhood. REFER-N finds a corresponding termination point, the closest example in $e_t \in E$ tagged with a different class label, referred to as the *point of class change*. The neighbourhood $E(e_s, e_t)$ then contains the set of training examples $\{e_1, e_2, ..., e_k\}$ such that $class(e_i) = class(e_j)$ for any $1 \leq i \leq k$ and $1 \leq j \leq k$ and $d(e_s, e_i) \leq d(e_s, e_t)$ for eny $1 \leq i \leq k$, where $d(e, e')$ is the Hamming distance between two examples e and e'. The partitioning of the example space proceeds in $E \backslash E(e_s, e_t)$ by considering the previous point of class e_t change as the new starting point and the process is repeated until the entire set of training examples is allocated a partition. Figure 5.2 illustrates this process. Since the space of binary strings is difficult to represent in a figure, we draw an analogue with two-dimensional space. Figure 5.2(a) depicts the process of neighbourhood construction starting with the example in the centre of the neighbourhood E_1. The heavy arrows show the progression of the starting point of each neighbourhood. Figure 5.2(b) shows the resulting comparisons made by REFER.



(a) Analogue of neighbourhood construction in two-dimensional Euclidean space.



(b) Comparison between neighbourhood-pairs of differing class

Figure 5.2: Neighbourhood construction and comparison in REFER

5.3.3 Modelling the typical REFER-N neighbourhood

There are several effects at work in REFER-N's neighbourhood construction which encourage coverage between features in individual neighbourhoods. However, in order to benefit the reduction of features, the conditions for combined coverage in section 5.3.1 need to be satisfied. Considering the properties of examples in an isolated neighbourhood, aiming to meet these conditions requires two patterns in the Boolean data. Firstly, six out of the seven conditions associated with partial coverage require constant features on at least one side of the comparison. Secondly, the features should exhibit coverage relationships among each other, since the remaining conditions all require this.

Consider a neighbourhood constructed by REFER-N. The starting, or seed, example is labelled e_s. There are n other examples in the neighbourhood, $n \geq 0$, labelled $e_1, ..., e_n$. The radius of the Hamming sphere that the neighbourhood represents is r. We define some necessary terms describing features.

Definition 5.13 (constant feature, variable feature, s-true, s-false). If a feature either has the value false for each example in a neighbourhood or true for each example in a neighbourhood, it is termed *constant*. Otherwise, it is termed *variable*. Where a feature is true for the starting example e_s, it is termed *s-true*, otherwise it is *s-false*.

An example may only differ from e_s in at most d features and due to the nature of the construction, its values do not depend on any other example than e_s. For a neighbourhood of n examples, there are at least $m - dn$ constant features, where m is the number of features in the dataset. Small neighbourhoods and datasets with many features therefore lead to many constant features, which cover each other. To a lesser extent, the Hamming-based clustering in REFER-N promotes coverage between particular kinds of features. Since every constant feature either covers or is covered by every feature, we restrict our attention mainly to coverage among variable features[6]. To aid the analysis, we characterise an example in terms of its difference from the starting

[6]The remaining change-sets $\{\}$, $\{\bar{f}g\}$ and $\{f\bar{g}\}$ refer to interactions between either two constant features (in the first case) or a constant

Feature		f	g	f	g	f	g	f	g
	e_s	0	0	0	1	1	0	1	1
Equal	$\bar{f}\bar{g}$	1	1	1	0	0	1	0	0
Exclusive	$\bar{f}g$	1	0	1	1	0	0	0	1
	$f\bar{g}$	0	1	0	0	1	1	1	0
Dependent	fg	1	0	1	1	0	0	0	1
	$\bar{f}\bar{g}$	1	1	1	0	0	1	0	0
Fully variable	$\bar{f}g$	1	0	1	1	0	0	0	1
	$f\bar{g}$	0	1	0	0	1	1	1	0
	$\bar{f}\bar{g}$	1	1	1	0	0	1	0	0

Table 5.2: Feature values included in a neighbourhood for f and g for each class of change-set.

example e_s. By doing so, we aim to determine the likelyhood that requisite conditions for coverage are attained.

Definition 5.14 (change-notation for an example and a pair of features, change-set for a neighbourhood). Given a seed example e_s, we characterise each example e in the neighbourhood of e_s with respect to features f and g by expressing the *change* between e and e_s as a symbol in the set $\{fg, \bar{f}g, f\bar{g}, \bar{f}\bar{g}\}$. The absence of a bar over f (or g) indicates the values for f (or g) do not change between e and e_s. A bar indicates a change (inversion of the bit). Furthermore, given a neighbourhood of examples E_i and a pair of features (f, g), the *change-set* is the set of changes appearing for at least one example in E_i.

The change-sets are immediately useful because they characterise where the coverage conditions hold. We may demonstrate the utility or otherwise of a partitioning method by the incidence of feature-pair classes, defined by their change-sets. Recall that we consider only feature-pairs which are both variable, since where one is constant, it is necessarily covered or covers another. Furthermore, we omit the change fg from change-sets with no loss of generality, since it is the case that the presence or absence of an example with change fg makes no difference to the coverage between features f and g. This framework introduces the following classes as subsets of $\{f\bar{g}, \bar{f}g, \bar{f}\bar{g}\}$. *Equal* feature-pairs only ever change together in the neighbourhood. Their change-set is $\{\bar{f}\bar{g}\}$. *Exclusive* feature-pairs exhibit changes only in f or g and have a change-set $\{\bar{f}g, f\bar{g}\}$. *Dependent* feature-pairs are such that where g changes, f does also. The converse is not true; f may change independent of g. The change set is $\{f\bar{g}, \bar{f}\bar{g}\}$. *Fully variable* feature-sets represent all possible changes, with a change-set of $\{\bar{f}g, f\bar{g}, \bar{f}\bar{g}\}$. These cases can be shown applied to different combinations of values for f and g in e_s in table 5.2. Later, we consider resulting coverage relationships between these features. Recall that REFER-R considers true values against false values for the first neighbourhood in a neighbourhood-pair comparison, and the converse for the second neighbourhood. While we consider the former in our coverage relationships, equivalent categories can be obtained for coverage among values for the value false.

5.3.4 The typical REFER-N neighbourhood

In order to understand the effect of this partitioning approach on the frequency of these cases in the data, we model a typical neighbourhood as follows: A neighbourhood of n examples each consisting of m features differs from its starting example e_s in d features per example. We assume the changes are independently and randomly distributed over each example. For a pair of (variable or constant) features (f, g), the probability that

feature and a variable feature (in the second case).

neither f nor g changes from e_s is therefore $P(fg) = \frac{(m-d)^2}{m^2}$. The probability that one (either f or g) changes is $P(\bar{f}g) = P(f\bar{g}) = \frac{d(m-d)}{m^2}$. The probability that both a and b changes is $P(\bar{f}\bar{g}) = \frac{d^2}{m^2}$. We model these four cases $\{fg, \bar{f}g, f\bar{g}, \bar{f}\bar{g}\}$ as independent states which an example can take with respect to two features f and g. Additionally, for any example and feature-pair f and g, the example must take one of these states. With respect to feature coverage, the existence of fg in a neighbourhood leaves its feature coverage relationships invariant.

For equal feature-pairs (change-set $\{\bar{f}\bar{g}\}$), coverage only occurs where the features f and g remain equal in each example. Equivalently, f and g must be both s-true or both s-false. By rearranging and applying the Binomial Theorem[7], in a neighbourhood of n examples, the probability of f only ever changing with g is as follows:

$$\begin{aligned} P(\text{no } f\bar{g}, \text{no } \bar{f}g, \geq 1 \bar{f}\bar{g}) &= \sum_{i=1}^{n} \binom{n}{i} P(\bar{f}\bar{g})^i (1 - P(\bar{f}g) - P(f\bar{g}) - P(\bar{f}\bar{g}))^{n-i} \\ &= \sum_{i=1}^{n} \binom{n}{i} P(\bar{f}\bar{g})^i P(fg)^{n-i} \\ &= (P(\bar{f}\bar{g}) + P(fg))^n - P(fg)^n \end{aligned}$$

For exclusive feature-pairs (change-set $\{\bar{f}g, f\bar{g}\}$), coverage only occurs where one feature is s-true and the other s-false. The s-true feature f covers the s-false feature g iff f and g do not change for any examples in the neighbourhood. In a neighbourhood of n examples, the probability of having at least one example changing in f but not g, and at least one changing in g and not f, and no examples in which both change, is calculated as follows, assuming $P(f\bar{g}) = P(\bar{f}g)$.

$$\begin{aligned} P(\geq 1 f\bar{g}, \geq 1 \bar{f}g, \text{no } \bar{f}\bar{g}) &= 1 - P(\geq 1 \bar{f}\bar{g}) - P(\text{no } \bar{f}\bar{g}, \text{no } \bar{f}g, \text{no } f\bar{g}) \\ &\quad - P(\text{no } \bar{f}\bar{g}, \geq 1 \bar{f}g, \text{no } f\bar{g}) - P(\text{no } \bar{f}\bar{g}, \text{no } \bar{f}g, \geq 1 f\bar{g}) \\ &= (1 - P(\bar{f}\bar{g}))^n - P(fg)^n - (P(\bar{f}g) + P(fg))^n + P(fg)^n \\ &\quad - (P(f\bar{g}) + P(fg))^n + P(fg)^n \\ &= (1 - P(\bar{f}\bar{g}))^n - P(fg)^n - 2(P(\bar{f}g) + P(fg))^n + 2P(fg)^n \end{aligned}$$

For dependent feature-pairs (change-set $\{f\bar{g}, \bar{f}\bar{g}\}$), s-true features may cover other s-true features, and s-false features cover other s-false features. In s-false features, f covers g where g does not change in any example without f changing. Similarly, between s-true features, g covers f where g does not change in any example without f changing. In a neighbourhood of n examples, the probability of having at least one example change in f and none in g, or equivalently under variable renaming, one example change in g and none in f, but in both cases at least one example change in both f and g, is calculated similarly to exclusive feature-pairs:

[7] Since $(x+y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i} = \sum_{i=1}^{n} \binom{n}{i} x^i y^{n-i} + \binom{n}{0} x^0 y^n = \sum_{i=1}^{n} \binom{n}{i} x^i y^{n-i} + y^n$, it is true that $(x+y)^n - y^n = \sum_{i=1}^{n} \binom{n}{i} x^i y^{n-i}$

$$\begin{aligned}
P(\geq 1 f\bar{g}, \text{no } \bar{f}g, \geq 1 \bar{f}\bar{g}) &= 1 - P(\geq 1 \bar{f}g) - P(\text{no } \bar{f}g, \text{no } \bar{f}\bar{g}, \text{no } f\bar{g}) \\
&\quad - P(\text{no } \bar{f}g, \geq 1 \bar{f}\bar{g}, \text{no } f\bar{g}) - P(\text{no } \bar{f}g, \text{no } \bar{f}\bar{g}, \geq 1 f\bar{g}) \\
&= (1 - P(\bar{f}g))^n - P(fg)^n - (P(f\bar{g}) + P(fg))^n + P(fg)^n \\
&\quad - (P(\bar{f}\bar{g}) + P(fg))^n + P(fg)^n \\
&= (1 - P(\bar{f}g))^n + P(fg)^n - (P(\bar{f}g) + P(fg))^n - (P(\bar{f}\bar{g}) + P(fg))^n
\end{aligned}$$

For fully variable feature-pairs (change-set $\{\bar{f}g, f\bar{g}, \bar{f}\bar{g}\}$), no coverage between the two features f and g is possible, since all true/false combinations exist within the neighbourhood for f and g. The probability of there being at least one example for each of the following cases: (i) the example changes in f but not g (ii) the example changes in g but not f (iii) the example changes in both f and g, is as follows. Here, $P(\geq 1 \bar{f}\bar{g}, \text{no } f\bar{g})$ is calculated similarly to the equal feature-pair case, and $P(\geq 1 \bar{f}\bar{g}, \geq 1 f\bar{g}, \text{no } \bar{f}g)$ follows from the dependant feature-pair case. We combining these and cancelling for $P(f\bar{g}) = P(\bar{f}g)$ and use the fact that $(1 - P(\bar{f}g))^n$ being equivalent to $(P(\bar{f}\bar{g}) + P(fg) + P(\bar{f}g))$.

$$\begin{aligned}
P(\geq 1 f\bar{g}, \geq 1 \bar{f}g, \geq 1 \bar{f}\bar{g}) &= 1 - P(\text{no } \bar{f}\bar{g}) - P(\geq 1 \bar{f}\bar{g}, \text{no } f\bar{g}) \\
&\quad - P(\geq 1 \bar{f}\bar{g}, \geq 1 f\bar{g}, \text{no } \bar{f}g) \\
P(\geq 1 f\bar{g}, \geq 1 \bar{f}g, \geq 1 \bar{f}\bar{g}) &= 1 - 2(1 - P(\bar{f}g))^n - (1 - P(\bar{f}\bar{g}))^n \\
&\quad + 2(P(\bar{f}g) + P(fg))^n + (P(\bar{f}\bar{g}) + P(fg))^n - P(fg)^n
\end{aligned}$$

We now have expressions giving probabilities for each case which, in some situations, does not form a coverage relationship with the constant features of either type, i.e. either those features which are constantly true or constantly false for each example in a neighbourhood. By substituting the expressions in terms of m (the number of features in the dataset) and d (the number of differences from e_s) for the variables $P(fg)$, $P(\bar{f}g)$, $P(f\bar{g})$ and $P(\bar{f}\bar{g})$, we can determine the probabilities in terms of m, d and n (the size of the neighbourhood). Since these terms contain complex powers of differing sums and are not amenable to further simplification, we analyse the results of REFER-R in terms of these probabilities by calculating them for varying m, d and n. We summarise the effects of each kind of example in terms of the combination of possible changes $\bar{f}g$, $f\bar{g}$ and $\bar{f}\bar{g}$ in table 5.3. Since any combination of bits can cover a constant feature, we consider only the interaction between two variable features f and g. Again, only coverage for truth-values is considered; analogous coverage of falsehood-values exists if the necessary subset relationships for truth-values exist.

Table 5.3 demonstrates a number of important results of change-sets in neighbourhoods and the effects on coverage. The notation $f > g$ means f covers g. Observe that where a constant feature is involved, coverage exists for all values of (f,g), though usually in one direction only. Furthermore, where two variable features are involved, coverage only exists in half the possible values for (f,g), for variable feature-pairs which are not fully variable. In order to appreciate the behaviour of REFER-N, we consider the incidence of these key categories of feature-pairs, using the probabilities derived.

Change present				value of (f,g) for e_s			
$\bar{f}g$	$f\bar{g}$	$\bar{f}\bar{g}$		(0,0)	(0,1)	(1,0)	(1,1)
absent	absent	absent	Constant features	$f=g$	$f<g$	$f>g$	$f=g$
absent	absent	present	Equal feature-pair	$f=g$	—	—	$f=g$
absent	present	absent	Variable vs. constant	$f<g$	$f<g$	$f>g$	$f>g$
absent	present	present	Dependent feature-pair (equiv)	$f<g$	—	—	$f>g$
present	absent	absent	Variable vs. constant	$f>g$	$f<g$	$f>g$	$f<g$
present	absent	present	Dependent feature-pair	$f<g$	—	—	$f>g$
present	present	absent	Exclusive feature-pair	—	$f<g$	$f>g$	—
present	present	present	Fully variable	—	—	—	—

Table 5.3: Effects on coverage of changing values in features f and g over a neighbourhood of examples.

5.3.5 Relationships among probabilities

Considering all feature-pairs in all neighbourhoods for a particular decomposition, the frequency of the various cases of variable feature-pairs of of immediate interest in estimating to what extent the conditions for coverage are fulfilled by the features. The eight cases in table 5.3 correspond to different levels of the subset coverage condition in either side of a comparison undertaken by REFER-R. Where both features in a feature-pair are variable, coverage exists in only half the possible values for (f,g). We are therefore interested in minimising these. Furthermore, the fully variable case stops any coverage, and should be even more discouraged.

We visualise how frequently these cases occur with respect to each other using the expressions for the typical REFER-N neighbourhoods derived in section 5.3.4, using three-dimensional plots comparing relative incidence of the cases for various values of m, d and n (5, 20 and 50). Figures 5.3 and 5.4 demonstrate the incidence of the fully-variable feature-pairs in the typical neighbourhood, since these preclude any coverage between features. Figure 5.3 shows the number of variable feature-pairs appearing for each fully-variable feature pair and figure 5.4 the number of all feature-pairs for each fully-variable feature pair. These allow us to understand the incidence of fully-variable feature pairs in a typical neighbourhood. It is desirable that these are low, since they are the most coverage-breaking. To a lesser extent (half the possible cases of (f,g) in e_s, we also wish to reduce the incidence of variable feature-pairs and variable features generally. The remaining figures 5.5 and 5.6 consider the proportions of variable feature-pairs. Figure 5.5 shows the proportion of feature-*pairs* which are variable, while figure 5.6 shows the proportion of *features* which are variable.

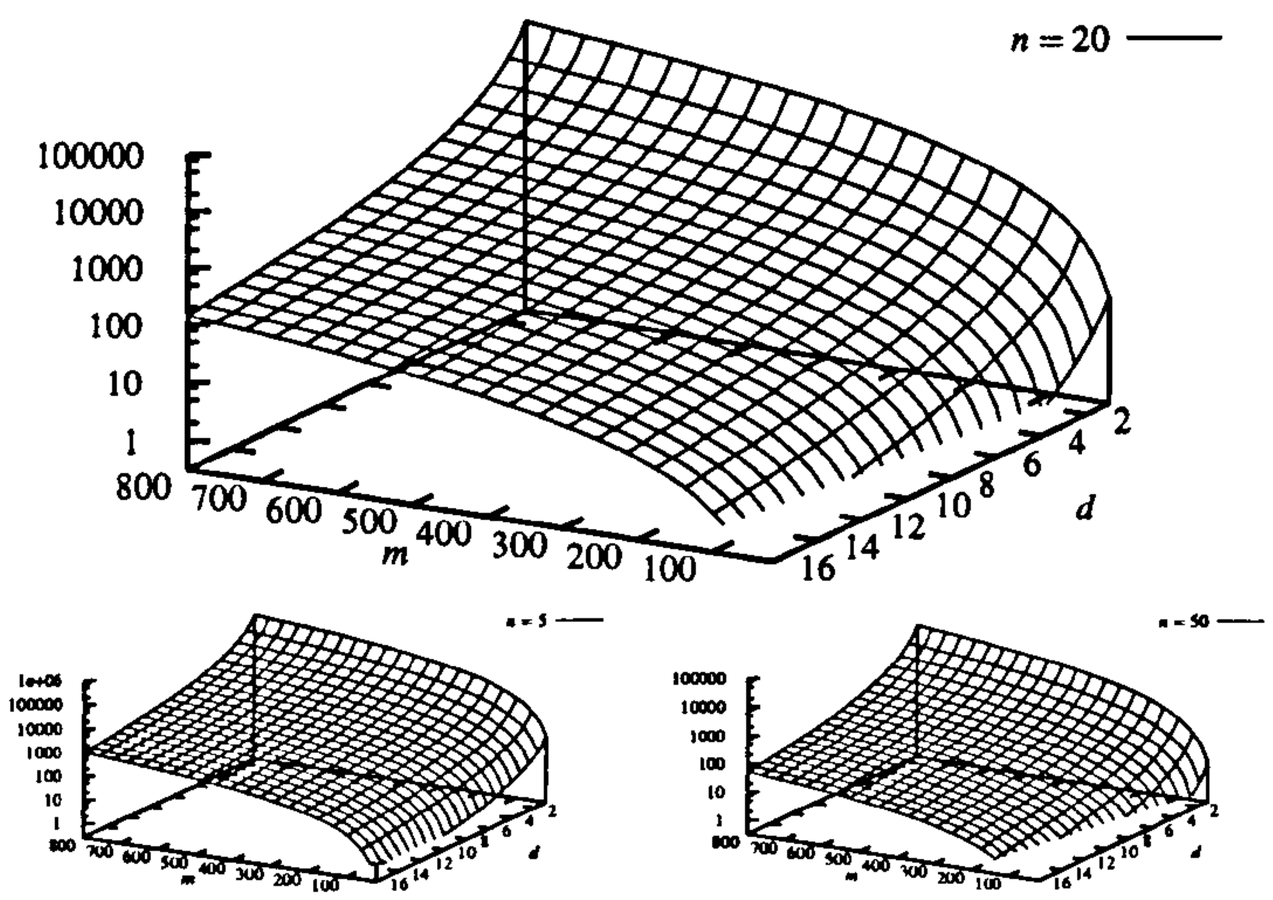


Figure 5.3: Incidence of variable feature-pairs appearing for each fully-variable feature-pair.

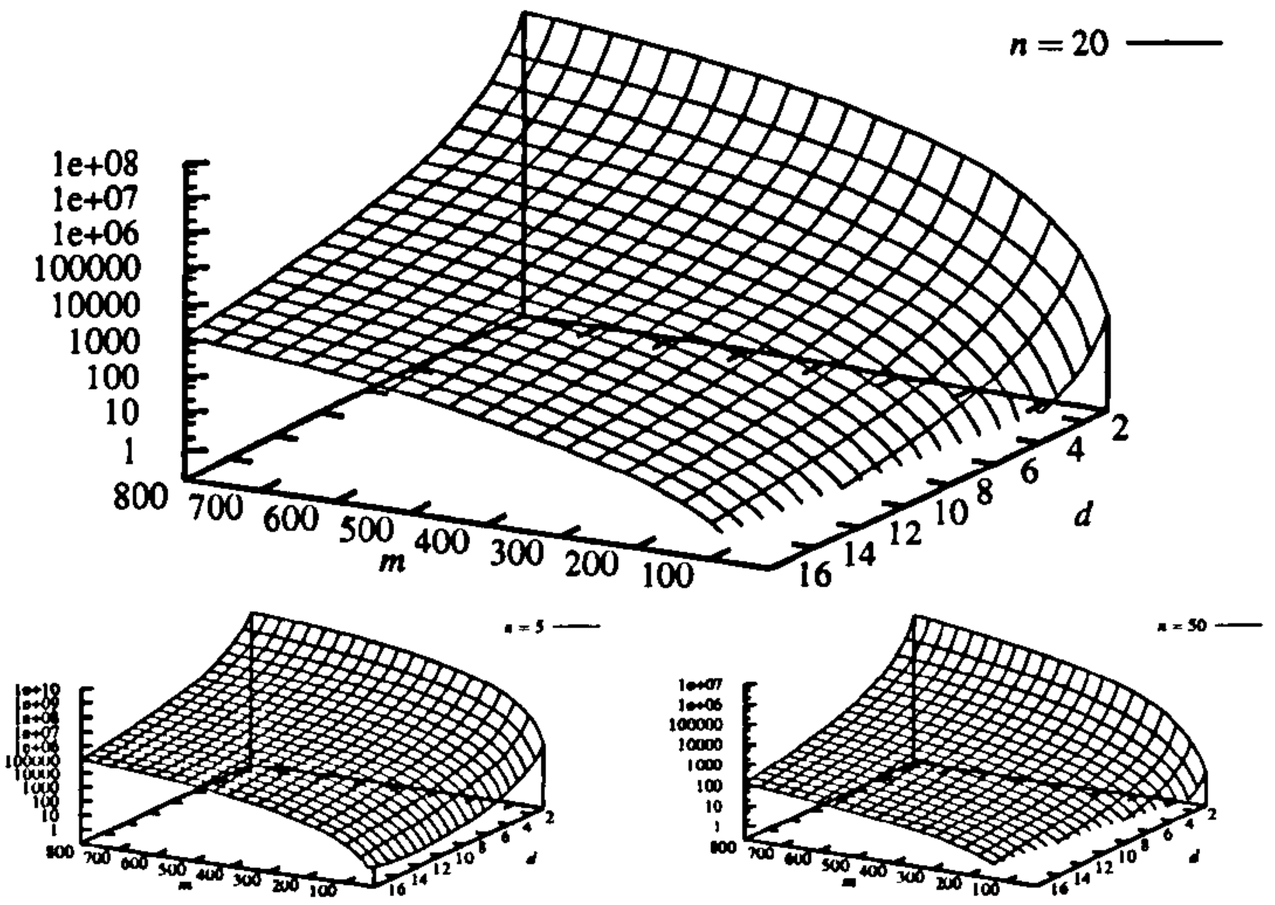


Figure 5.4: Incidence of all feature-pairs appearing for each fully-variable feature-pair.

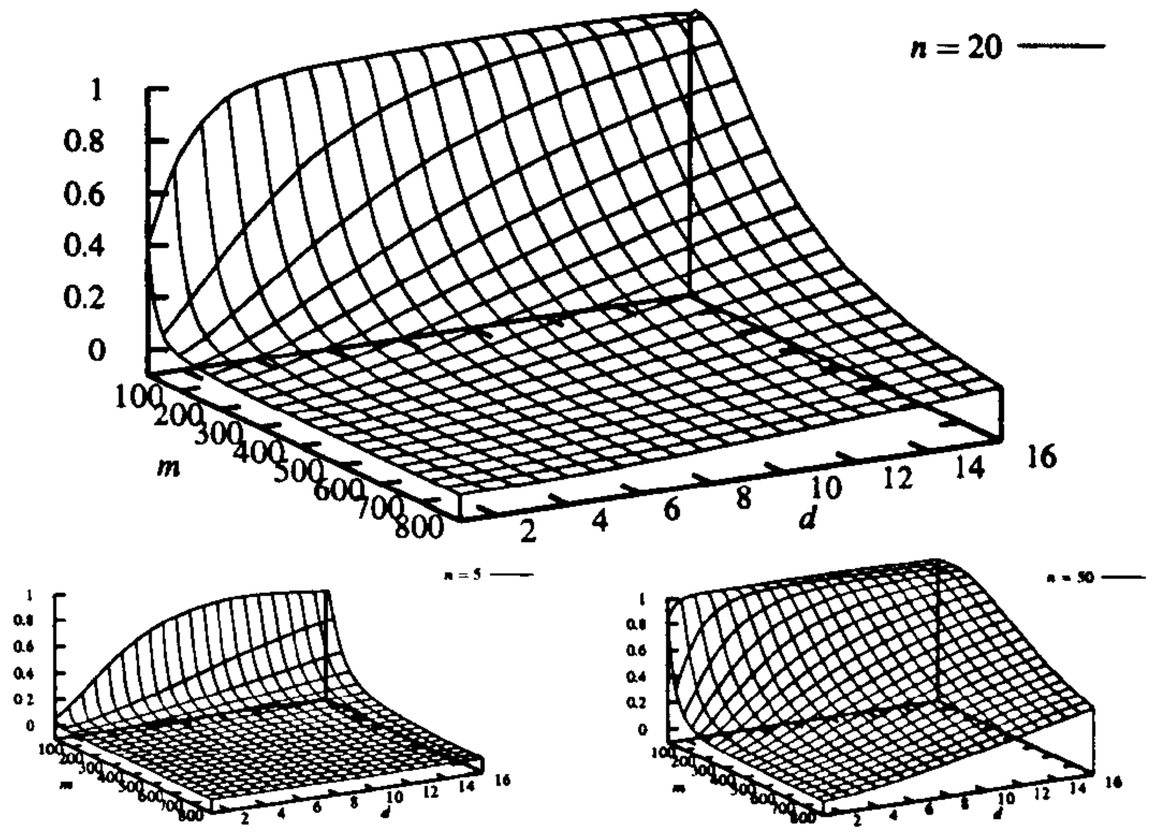


Figure 5.5: Proportion of feature-pairs which are variable.

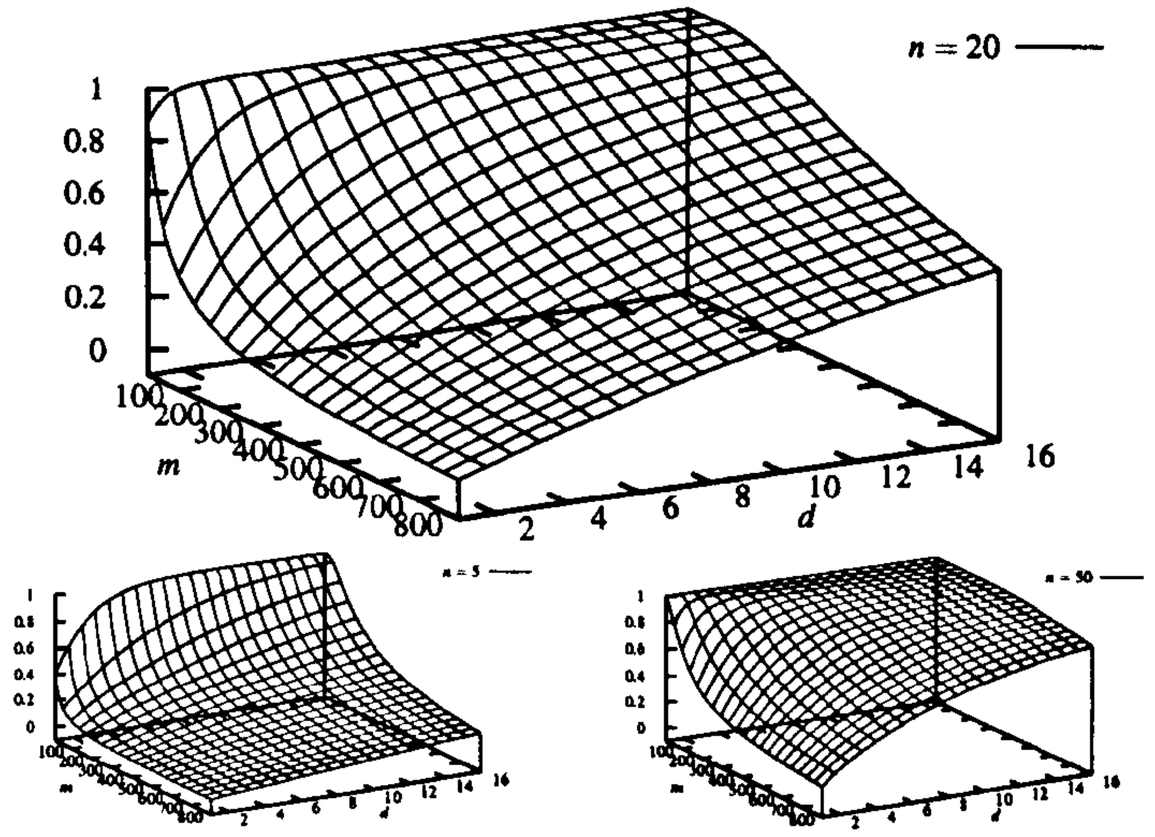


Figure 5.6: Proportion of features which are variable.

The data shows a number of important effects which show the utility of Hamming distance as a neighbourhood decomposition approach, as used in REFER-N.

Figure 5.3 suggests that the incidence of fully-variable feature-pairs increases steadily with increasing d and decreases with decreasing m. Generally, fully-variable feature pairs occur rarely in data as it has been modelled here, especially where neighbourhoods are very small. Even for low values such as $n = 3$, $d = 5$, $m = 400$, fully-variable feature-pairs exist only around 1 in 10,000 times in the data. With increasing n, feature-pairs become steadily more common, although even at $n = 20$, equal, equivalent and dependent feature-pairs are still approximately 10^3 more likely than fully-dependent ones, and tend to occur one in 10^4 times for the value of these parameters. The low numbers of fully-dependent feature-pairs mean that, in general, feature-pairs are either described by a case in which coverage exists for half of the possible values of (f, g) in e_s, or for which coverage is preserved regardless of these values. Since more coverage occurs, there are more coverage

relationships between features, and as a result, more features can be removed by REFER-R overall. Figure 5.4 demonstrates that when compared against all (non-fully-variable) possible classifications of feature-pairs, the effect is even more pronounced (see the z-axis), even though the shape of the surface is approximately the same.

Turning to figure 5.5, for lower n, feature-pairs do not tend to be covered by the four cases above (equal, dependent, exclusive and fully variable), except for lower values of m and to a lesser extent, increasing d. The cases become much more prevalent in very large neighbourhoods — in large neighbourhoods, most feature-pairs appear to be variable for even small d. If $n = 30$ and $d = 8$, 90% of feature pairs are variable for $m < 250$. Comparing with figure 5.6, except for low m, all the cases become steadily more likely as n increases, and the same is true for the number of variable feature-pairs. At the parameters given, approximately half the feature-pairs are variable, of which one tenth are likely to fall into one of the cases given if n is as high as 30.

Even though this demonstrates that even in large neighbourhoods, the nature of REFER-N avoids the incidence of feature-pairs containing a variable feature, REFER-N tends to produce a variety of different-sized neighbourhoods, many of which are small. In these neighbourhoods, the incidence of the four cases, and in particular the case of a fully-variable feature-pair, is far smaller. A possibly contributory factor is likely to be that points of class change delimit small hyperspheres, and furthermore, the iterative traversal of the algorithm from point of class change to point of class change causes nearby data to be typically split up into several neighbourhoods, represented by Boolean hyperspheres which could be said to overlap in an m-dimensional Boolean space. As a result, neighbourhoods are typically small (low n) and correspond to (parts of) Boolean hyperspheres with small radii (low d).

Referring to the plots produced, these conditions are particularly important for encouraging feature coverage. The incidence of fully-variable feature pairs is very small indeed for all values of m, and the cases described in general are also very unlikely in general. The remaining feature-pairs produce coverage relationships in at least one direction for all values of f and g for e_s as a result.

5.4 Conclusion

This chapter has considered a combined approach to induction consisting of combining *propositionalisation*, which transforms the set of features found during a usually partial search of the hypothesis space into a Boolean dataset, and *feature reduction*, which aims to reduce the number of features in the transformed dataset such that a the learnability of a rule in the reduced dataset is preserved.

With regard to propositionalisation, we considered the particular task of query-based transformation, consisting of two separate steps, a process of feature construction followed by the evaluation of each feature using an evaluation function. Several forms of evaluation function were considered, primarily those belonging to existential and counting transformations. The decoupling of feature construction and evaluation removes the possibility of coverage-based pruning, and so language-based constraints as well as appropriate search bounds are of increased importance. We reviewed general approaches to feature construction, with particular emphasis on the individual centred representation, and studied approaches to filtering features. The section surveyed the range of work undertaken on propositionalisation techniques and systems, and considered its application to induction in the object model.

Propositionalisation often leads to the production of many redundant features, and removing them can lead to improvements in predictive performance, induced theories of better quality, and reduced induction time. We have also presented and analysed some *feature reduction methods* for preprocessing propositionalised datasets

in order to remove those features which are logically redundant for classification rule learning. Of particular interest in this chapter has been feature removal by the partitioning of the example set into neighbourhoods of the same class and comparing the features for logical coverage. It was shown that reducing the data this way would not compromise the ability of an arbitrary classification rule learner to learn a complete and consistent hypothesis. It has been shown empirically (in [5]) that many more features may be reduced using partitioning methods than by comparing the dataset as a whole due to the possibility of different features in each neighbourhood working together to cover another. The conditions under which this effect occurs, known as combined coverage, were derived. A fast method based on clustering by Hamming distance — REFER-N — was introduced and a probabilistic analysis demonstrated that it promotes coverage conditions in the neighbourhoods it produces. In section 7.5.3, later in this thesis, we perform an empirical study of REFER on propositionalised data.

Chapter 6

The CoSINUS system

In chapters 2 and 3 we described CORLOG, a logical language for reasoning in domains which are naturally expressible in the object-oriented data model. In chapter 4 we presented an approach to refinment and induction in this language. In chapter 5 we introduced the framework of propositionalisation and a general method for reducing the dimensionality of the highly-dimensional multi-class Boolean datasets produced during this approach. This chapter describes a system, called COSINUS, which implements this approach to refinement by propositionalising using features searched using the refinement operator. The approach thus transforms a complex inductive logic programming approach to an attribute-value form suitable for learning by an existing propositional learner. Rules are repeatedly learnt in a *covering loop* in order to learn whole theories — disjunctions or rules — from object databases. The feature reduction approach discussed in chapter 5 is then applied to detect (further) redundancy among generated clause queries and to reduce the dimensionality of the learning problem.

The chapter is structured as follows. In section 6.1 we introduce the basic three-tiered structure of the learner, which serves to categorise the constituent algorithms in in the approach and gives a conceptual overview of the COSINUS system. Section 6.2 presents the syntax of the key learning parameters and metaknowledge which are specified to the system and considers the way in which aspects of a CORLOG feature are represented in the logic programming language adopted and how the metaknowledge governing their validity is checked. Section 6.3 considers the implementation of the search at the feature level, whereas section 6.4 considers the implementation of the propositionalisation subsystem which generates rules and theories from these features. Section 6.5 concludes.

6.1 From feature construction to theory induction

So far we have considered *feature* search over the space of CORLOG clauses. Features alone represent only the building blocks of sets of rules which are used to classify individuals. As well as discussing the implementational aspects of the feature search, this chapter presents the method whereby these sets of rules are constructed, lifting induction from the feature level to rules, and sets of rules, known as theories. These theories are learnt by COSINUS using an iterative, hierarchical approach based on propositionalisation. As discussed in chapter 5, propositionalisation systems work by transforming a first-order (multi-relational), individual-centred problem into a propositional (single-relational) form. The propositionalisation process generates rules from features.

Single mode generates the full rule set in one single propositionalisation, whereas *iterative mode* generates them one-by-one, progressively generating a feature with each new iteration.

This hierarchy gives rise to a learning approach which takes place on several levels. Just as features are constructed by a search through a space of features, we can draw an analogy by considering the search taken by the propositional learner through a space of rules, and the search by the covering algorithm through the space of theories. In order to motivate this analogy, we recall that query-based propositionalisation generates a set of features (queries) belonging to a carefully-constrained language $\mathcal{L}$, usually with respect to some quality or stopping criteria to avoid exhaustive generation of all possible features. Recall further that a feature set is a set of features which share the same interface in its head variables. Rules are conjunctions of these features. More formally,

Definition 6.1 (rule). Given a feature set $FS = \{f_1, f_2, ..., f_n\}$ under an interface int_{FS}, as defined in definition 5.4, a *rule* is a CORLOG clause of the form

$$\mathsf{O[class \rightarrow r]} \leftarrow \mathsf{O[f_{i_1}], O[f_{i_2}], ..., O[f_{i_m}]}. \tag{6.1}$$

where each $f_{i_j} \in FS$, $\forall 1 \leq j \leq m$. The head of the F-rule consists of a method which outputs r, a constant class label for the individual O.

In some propositional rule learners, the features appearing in a rule may be negated. Furthermore, the undecomposability of features assists in reducing the redundancy of this search. Since no two features may be expressed as the conjunction of another, there are fewer rules — conjunctions of features — which are equivalent to each other. Finally, we recall the definition of a theory as a disjunction of rules. The rule search may be demonstrated by considering a simple rule learner which searches a *rule body lattice*; as the set of features forms a generality lattice, the set of rule bodies forms a similar structure. Given a feature set FS, we define a lattice over the set of possible conjunctions — the powerset of features $\mathcal{P}(FS)$[1]. The lattice is ordered by a relation $\leq_R$, such that $C \leq_R D$ holds if the features appearing in the conjunction C are a subset of those in D. A rule body may then be refined under a refinement operator ρ_R by adding a feature to the conjunction. Furthermore, by defining some arbitrary total order $\leq_F$ over the set of features, we can refine in a non-redundant fashion by redefining the refinement operator such that $C' \in \rho_R(C)$ iff $C' = C \wedge F$, where $F \in S$ is a feature such that $\nexists G \in S$ such that $F \leq_F G$, effectively imposing a spanning tree on the rule refinement graph. By adopting appropriate rule quality measures and stopping criteria, the search may take place in an analogous way to feature search. In practice, propositional learners conduct searches through the rule space using more sophisticated techniques and using special-purpose heuristics.

The induction approach adopted in the approach presented in this thesis is a *three-tiered approach*. At the *feature level* — the lowest level — the system searches through a space of hypotheses to find a subset which cover the examples according to a quality measure. These are converted into a single table through a propositionalisation tranformation. A *feature set* is generated by a propositionalisation process. At the *rule level* — the intermediate level — a propositional learner selects a conjunction of the features to build a rule which describe a subset of the examples according to its own quality measure as defined in definition 6.1. At the *theory level* — the highest level — the system successively removes the examples covered by the discovered rules, and repeats the process on the remaining examples. Where a seed example or coverage testing set is used,

[1] This example considers only positive features but may be trivially extended to include negative features.

a new sample is taken from the set of remaining examples. The disjunction of the rules then forms the resulting theory.

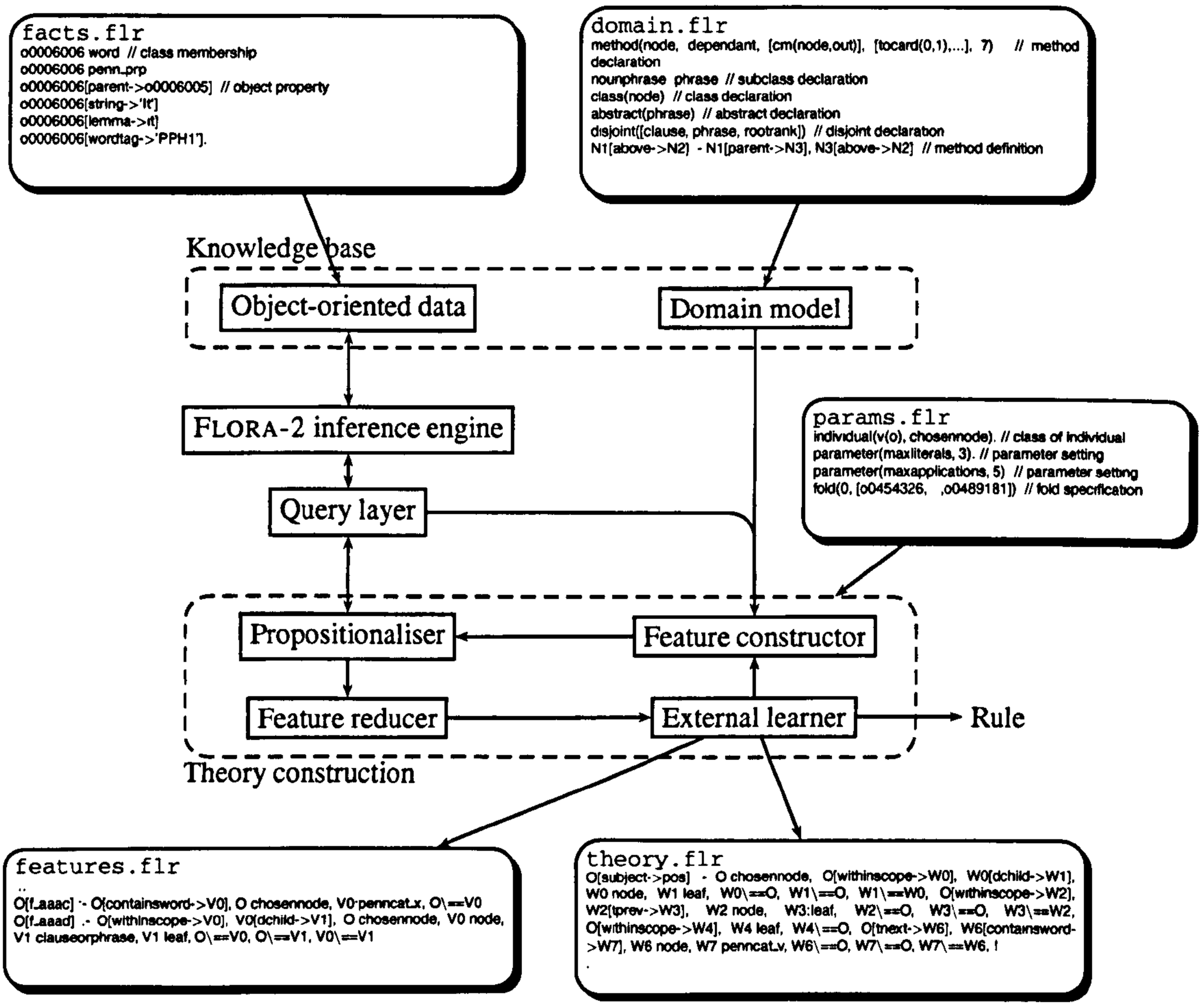
There are a number of benefits to adopting this hierarchical approach. Firstly, established propositional learners with sophisticated methods to determine the best rule may be employed. This may be more effective than using built-in heuristics and quality measures for rules in ILP learners. Secondly, the induced theories are naturally modularised and is therefore suited to the object domain. Thirdly, In some domains, the computationally-expensive search in ILP can be replaced with a less expensive search. When the first-order domain has been transformed (constructing the features), the propositional domain may be handled by efficient rule learners to produce often very complex features without requiring explicit enumeration in the ILP learner. Finally, approaches such as covering algorithms may be incorporated into the learning process by virtue of the way the propositional learner operates. For example, in an iterative approach, a rule learner may be used to select the best rule, the propositionaliser removing all examples covered by it, and repeating until no examples remain. Other possibilities for integration between the first-order ILP system and the cooperating propositionalisation system then present themselves.

6.1.1 System overview

At the system level, this three-tiered search is implemented in a cycle — a covering loop — of feature construction, propositionalisation, feature reduction and rule construction by an external learner. This cycle is the core of the operation of the CoSinus system. We present an overview of the system before highlighting key implementational features of it.

CoSinus was developed and implemented in several programming languages. The core implementation was in the XSB dialect [123] of Prolog. The Flora-2 [141] programming language was adopted for representation of the knowledge base and its deductive engine, as well as the representation of the domain knowledge and additional modules aiding efficiency and integration between Flora-2 and XSB. Flora-2 is itself implemented in XSB Prolog, acting as a translational layer between the object database and the inference engine of XSB. Flora-2 is a natural choice for our needs since it is based on the reasoning mechanisms and syntax of the F-Logic language.

A number of other systems exist which involve F-Logic for reasoning, and could therefore be adapted to CorLog. The one most similar to Flora-2 is FLORID [89], a deductive object-oriented database which uses F-Logic in its data definition language and as a query language, and is particularly suited to queries on web documents in XML, HTML and RDF formats. It is full-featured as an F-Logic query language, but lacks some programming constructs necessary for the construction of an ILP system which are used in CoSinus, most notably in the areas of meta-programming and Prolog-integration. Additionally, the suitability of representing ontological databases in F-Logic have led to a number of query languages for such databases. RDF knowledgebases, used in semantic web applications, describe classed objects with named properties in a series of subject-predicate-object statements. Classes of objects and their properties may form hierarchies. Systems such as Ontobroker [34] and its underlying F-logic inference engine SILRI, and TRIPLE [130] provide a query language for querying RDF and similar knowledgebases using F-logic. Again, while such languages often utilise the F-Logic syntax, they function primarily as *query*, rather than *programming*, languages. The CoSinus system can be broken down into a number of core components, or modules. Figure 6.1 shows the interoperation of these modules. We describe each briefly in turn, going into much more depth later in the



Boxes in the centre of the diagram represent modules of the CoSINUS system. Shadowed boxes represent important input and output files and contain example lines from the a domain similar to the natural language domain in chapter 7.

Figure 6.1: An overview of the CoSINUS system

chapter.

Conceptually, the the *knowledge base* for a domain consists of two components: the object-oriented data and the domain model. They are decoupled both conceptually and in the implementation, allowing the definition of domain libraries specific to particular real-world learning tasks. The object-oriented *data* are the database facts, defining objects in terms of class membership assertions and method expressions assigning objects to the objects' named properties. The *domain model* consists of the inheritance hierarchy, the type signatures, deductive rules defining the implementations of methods, and metaknowledge expanding on specific semantic relationships for particular methods and inheritance relationships. The *inference engine* resolves queries against the knowledge base. For the purposes of induction, it determines whether a constructed clause covers an example by resolving it against the database. Additionally, the inference engine is used to query the data model, also expressed in F-Logic.

The feature search uses the domain model to construct a set of features by recursive application of the refinement operator, starting with the empty feature. Features are refined in Prolog as part of the *feature constructor*. The *query layer* translates an internal representation of a generic object query into one that can be queried on the FLORA-2 knowledge base, executing the query from inside XSB-Prolog. The query layer provides a useful point of abstraction between the feature refiner and the inference engine, incorporating query transformations and optimisations.

For a constructed feature set, the *propositionaliser* determines for each feature and example whether the feature entails the example. As a result, a table of truth values are obtained. The *feature reducer* module applies the REFER feature reduction method described in chapter 5 to the propositionalised dataset. Features found to be redundant are removed from the feature set. The *external learner* module outputs the features detected non-redundant from the propositionalisation, and invokes the external, propositional learner. The learner outputs a set of rules describing the propositionalised examples. A subset of these rules are extracted and converted back into the form of an FLORA-2 query. Facilities for experimentation, such as testing accuracy over differing parameter settings in an n-fold cross-validation, are implemented over the whole theory construction loop.

6.2 Declaring metaknowledge and representing features

Discussion of object-oriented domain definition in previous chapters have presented an implementational syntax for object-oriented domain definition. A number of additional declarations are required by the learner in order to perform the feature search. They are defined as FLORA-2 facts, but adopt the more usual Prolog syntax.

Firstly, we consider the remaining declarations relating to the notion of class. The variable and class of the individual is declared by fact individual(v(o), c_i), which states that the class of the individual is c_i and the variable appearing in the head is O. The name of the target predicate is declared using a fact target(m), where m is the name of the method. Each simple class c to be used in refinement must be declared by a fact class(c). Families of parametric classes, meaning sets of legal bound elements, are defined by a combination of two predicates. pcb/1 is a list describing the bounds on a parametric class. The first element of the list is a term of the form param(P), denoting the bound is on parametric class P. The remaining elements describe the bound on each bound element of the parametric class. Where it is of the form exact(C), the element may only take the class C. Where it is of the form extends(C), the element may assume any class C' which is a subclass of, or equal to, C. pcm/2 defines subclassing across two parameteric classes, usually mapping arguments in one parametric class into arguments in its subclass. For example, the fact pcm(p_1(A, B), p_2(B, A, _)) would assert that $p_1(c_1, c_2)$ is a subclass of $p_2(c_2, c_1, c3)$. Where either kind of class is abstract, an additional fact abstract(c) is included. A set of mututally-disjoint classes $\{c_1, ..., c_n\}$ are represented by the fact disjoint($[c_1, ..., c_n]$). Finally, in order to correctly perform type specification, all classes must inherit from a common abstract superclass top. Classes without a superclass therefore are defined to subclass from top.

Next, we consider the remaining declarations relating to methods. Each method to be used in learning is given a method specification, defined by the fact method/5, and whose arguments are defined in definition 3.18. Recall from this definition that a method/5 class is of the form method(C, M, [cm(C_1, M_1), ..., cm(C_n, M_n)], T, K), where C is the target class, M is the method's name, for each argument C_i is the class of the argument and M_i its mode, T is a list of metaknowledge tags, and K is a unique number, Order-yielding methods are indicated by an additional fact, oym(C, M(C_1, ..., C_n), V_O, C_O) for a method M defined in class C, with input arguments of class C_i, and an output argument of C_O. The method is order-yielding given where the method returns V_O. For

example, oym(integer, lessthan(integer), true, bool) is an ordered method implementing the 'less than' order over integers through a method returning true where the target attribute is less than its argument.

Since the majority of the COSINUS system is implemented in XSB Prolog, the use of an intermediate representation for a feature as a Prolog structured term is adopted. A translational module mediates between the Prolog and FLORA-2 representations, constructing the equivalent query in the FLORA-2 syntax for resolution. This approach gives several advantages over a direct representation. Firstly, the Prolog representation may include other properties of the feature for the refinement process to use. Properties of the feature may easily be associated with it, most notably the class constraints on the terms in the feature. Secondly, the intermediate representation allows the convenient processing of the feature syntactically, permitting operations such as redundancy checking and class subsumption to be carried out without the need to perform resolution directly in FLORA-2.

Turning back to the first point, we briefly consider the representation of a feature, and in so doing, document the additional properties of a feature which are associated with it for the purposes of feature refinement. Some of these terms directly translate to elements of the equivalent FLORA-2 query whereas others serve to store properties of the feature during refinement. In the refinement algorithms below, a CORLOG feature is represented by a ternary Prolog term `rule(Head,Body,Meta)`. `Head` gives the class of individual. `Body` is a list of data expressions of the form $\mathtt{t_0 : m(t_1, t_2, ..., t_n)}$. Each $\mathtt{t_i}$ is either of the form `v(v)`, representing a variable with name V, or `c(t)`, representing the constant t. $\mathtt{t_0}$ correspondonds to the host object, $\mathtt{t_n}$ to the output argument and the remaining $\mathtt{t_i}$ terms correspond to input arguments. Finally, `m` is the the FLORA-2 method. The list thus translates into FLORA-2 molecules of the form $\mathtt{t_0[m(t_1, ..., t_{n-1}) \rightarrow t_n]}$.

`Meta` describes properties of the feature and its constituent literals and terms. `varinfo(V,C,M,Co)` asserts that variable V is constrained by a list of classes C — a class conjunction – and appears in arguments taking mode(s) M. The Co keeps track of the number of times the variable has been consumed, or bound with an earlier-occurring variable. $\mathtt{constinfo(T, C, P_L, P_A)}$ asserts that the constant T appearing in literal number P_L and argument number P_A is of class (conjunction) C. To ensure that the search does not exceed the user-specified bounds, `varcount(N)` asserts that there are N variables in the feature. `depth(N)` asserts that the feature is N refinement steps deep in the refinement lattice. A number of additional terms keep track of properties specific to refinement subprocesses. `odc/7` and `dfreshvars/1` maintain properties relating to valid downcasting operations on terms, whereas `freshvars/1` and `greatestmethod/1` relate to valid adding and waking operations. These are described in more depth in later discussion regarding the implementation of these processes.

Given the triple `rule`, an equivalent CORLOG expression may be easily generated. The `head` component is combined with the target predicate to form the head of the rule, while each element of the list in the `body` component may be transformed to an equivalent literal or method expression in the body of the corresponding CORLOG rule under the representation described above. Class membership constraints are then appended to the rule, one for each of the `varinfo/4` elements of the metadata; if a variable V is constrained to be a member of a class C by the metadata by appearing in the `varinfo/4` class list, a constraint $\mathtt{V : C}$ is appended to the translated rule. No such constraints are considered for constants, since they are both intrinsically classed and their constraints are not made more specific, except in the special case of an ordered method. Where object identity is adopted, additional constraints are introduced to ensure that each pair of terms in the feature are not equal. Using this approach, we may translate any rule in the representation described into CORLOG form, suitable for resolution by FLORA-2.

Checking the rules for type- and metaknowledge validity is undertaken syntactically on the internal Prolog

Algorithm 6.1: COSINUS: Top-level object-oriented induction algorithm

input : D, a FLORA-2 knowledge base.
input : $TM = C_I[M \Rightarrow C_P]$, a method signature representing the target method. C_I is the class of the individual, M a method defined on C_I, and C_P a class of possible predictions.
input : $Q(f, D)$, a feature quality criterion.
output: T, a set of rules predicting the value of M for new objects of class C_I

```
T ← ∅
E' ← {e|e : C_I}
TI_D ←PREPROCESS (D)
while |E'| > 0 do
    E ← seedexamples positive examples sampled uniformly from E'
    F_0 ← the empty feature body, with a single variable O of class C_I
    FS ←CONSTRUCTFEATURES (F_0, E, Q, D, TI_D)
    P_FS ←PROPOSITIONALISE (FS, TM, D)
    P'_FS ←REFER (FS)
    T' ←LEARNRULE (Pr'_FS)
    T ← T ∪ T'
    remove from E' all examples correctly covered by a clause in T
return T
```

representation described. We describe this process in more depth in section 6.3, which describes the feature search.

6.3 Implementing the feature search

The top-level COSINUS algorithm is shown in algorithm 6.1. The approach takes the form of a covering algorithm, iteratively searching for a feature to cover a subset of the examples in the database. The algorithm searches for a rule body which predicts a given value for a (one-to-one cardinality) target method. Prior to running, PREPROCESS computes initial type data for use over the series of runs which the algorithm takes. Inside the covering loop, a subset E of the examples of a given size is selected. A search of the feature space, using an implementation of the refinement operator described in chapter 4, to generate a feature set such that each feature is valid (with respect to the consumption of its variables and metaknowledge constraints), within feature generation bounds, and meets the quality criterion Q. Owing to the individual-centred representation, the initial feature has the empty body and is associated with a single variable of the class of individual, *i.e.* sharing a clause head of the form $\mathsf{O[M \rightarrow X]}$. This search is implemented in a depth-first fashion by the CONSTRUCTFEATURES algorithm. Once the feature set has been generated, PROPOSITIONALISE constructs a propositionalisation from the feature set FS against the database D with respect to the target method TM. REFER performs a feature reduction as described in algorithms 5.2 and 5.1. LEARNRULE runs an external learner, selecting a rule to cover E, and translating it back into a new part of the theory induced by COSINUS. The examples (correctly) covered are removed and the iteration loop continues until no examples remain. At this stage the theory is returned to the user.

Having presented the algorithm generally, we now consider each of the constituent processes of COSINUS in detail.

6.3.1 Preprocessing for feature set generation and COLLECTVALUES

In order to reduce the time complexity of the COSINUS learner, a number of preprocessing steps are undertaken. The three principal sets of knowledge which are computed are the set of objects belonging to subprimitive classes, additional structure over the subclass hierarchy, and the establishment of samples for testing coverage.

Primitive classes are the most basic ones in the system. In FLORA-2 they are the classes integer, float, number and symbol. In constructing features, we use these primitive types to determine whether it makes sense for a single object in one of these classes to be substituted as a constant for a variable of a compatible class. An object of any of these classes, or a subclass of any of these classes, is said to be *subprimitive*. Non-subprimitive classes can also be treated in the same way by introducing a fact cltype(c) for a class c. The preprocessing step gathers the set of values for each subprimitive type or cl-type and asserts a Prolog fact `type(C,SP,V)` for the class C, the primitive type(s) it inherits from SP, and a list of values V. If the size of the list of values V is equal to or less than the parameter setting `largetype`, it is then treated as a subprimitive class. We will continue discussion of the uses of subprimitive types later when we discuss substitution.

Secondly, the set of all possible classes in the knowledgebase are collected. We identify two types of class, simple and parametric classes. Firstly, all simple classes, declared by the facts class(c) are collected. From these, the metaknowledge describing bounds on parametric classes are used to calculate *all possible* valid parametric classes. These are then *topologically sorted*, *i.e.* sorted in such a way that for any pair of classes (c, c') such that c' is a superclass of c, c' appears earlier in the resulting list than c. The topologically-sorted list L is then asserted as classlist(L). The list introduces an intrinsic order over the classes, which is exploited later in the type specialisation mechanism.

The final form of precomputation aims to overcome the computational expense of the coverage test of a feature. Since it is time-consuming to determine the coverage of a feature over all possible examples, before each iteration of the learner, a sample of the examples is taken, and coverage is tested against these examples only. While this reduces the accuracy of the coverage test, it also reduces the time complexity of a run of COSINUS greatly.

6.3.2 Implementing feature generation and evaluation

CONSTRUCTFEATURES, presented in algorithm 6.2, is the central process of the COSINUS learner, We discuss its operation in general, considering the limits imposed on the feature search, and then consider the implementation of the processes performing the core refinement processes of performing a substitution, adding a new literal, and specialising the type of a term in the feature.

Feature search takes place by recursively calling the predicate `refine`, which simply takes a feature as input and produces a new feature as output. Recursion is bounded by depth; the `depth(N)` element of the metadata being incremented on each refinement. If this depth exceeds the parameter setting `maxapplications`, no further refinement is carried out. In general, we adopt example coverage as the quality criterion $Q(f, E)$, in which the number of examples in E which are correctly covered by f is compared with the setting `mincoverage`. If the number of examples covered is less than `mincoverage`, the feature is considered too specific and therefore invalid, meaning Q is false. Since metaknowledge-validity represent a set of criteria which all features must agree with, it could be argued that our Q is, in effect, a combination of the coverage criteria and metaknowledge validity. Evaluating Q may be used to prune away large parts of the search space, provided it possesses monotonicity properties, discussed in section 4.1.2. The minimum absolute frequency constraint

Algorithm 6.2: CONSTRUCTFEATURES: feature set search in COSINUS

input : f, an initial feature.
input : E, a set of examples used in testing quality.
input : $Q(f,E)$, a quality criterion testing the quality of a feature f against E
input : D, the object-oriented database
input : TI_D, type information for D
output: FS, a feature set consisting of rule bodies

if *depth of* $f \geq$ *maxapplications* **then return** $\emptyset$
$FS \leftarrow \emptyset$
$FS' \leftarrow$REFINE (f,D,TI_D)
foreach $f \in FS'$ *such that* $Q(f,E)$ **do**
  if f *is complete* **then** $FS \leftarrow FS \cup f$
  if f *is incomplete* **then**
    $FS \leftarrow FS \cup$CONSTRUCTFEATURES (f,E,Q,D,TI_D)
return FS

Algorithm 6.3: REFINE: feature set search in COSINUS

input : f, an initial feature.
input : D, the object-oriented database
input : TI_D, type information for D
output: FS, a feature set consisting of rule bodies

$FV \leftarrow$ fresh variables from f; $AV \leftarrow$ variables from f
$FS_s \leftarrow$ all $f\{V/V'\}$ for $V \in FV, V' \in AV$ such that classes of V,V' are compatible
$FS_o \leftarrow$ all order-deeping of data expressions in f taking some $V \in FV$ as its host variable
if *number of literals in* $f <$ *maxliterals* **then**
  $FS_a \leftarrow$ all features $f \cup f'$ such that f' is a data expression taking some suitable $V \in FV$ as its host variable, $V_i \in AV$ as inputs and a new variable as output
$FS_t \leftarrow$ all f' such that $f' =$DOWNCAST (f)
$FS = FS_s \cup FS_o \cup FS_a \cup FS_t$; $FS' = \emptyset$
foreach $f' \in FS$ **do**
  $valid \leftarrow true$
  if f' *is not multiplicity respecting* **then** $valid \leftarrow false$
  if f' *is not distinct-binding respecting* **then** $valid \leftarrow false$
  if f' *contains duplicate literals* **then** $valid \leftarrow false$
  if $valid = true$ **then** $FS' \leftarrow FS' \cup \{f\}$
return FS'

adopted here is anti-monotonic and therefore where a candidate feature is false for Q, its refinements may also be disregarded.

Algorithm 6.3 shows the REFINE process in summary. It attempts to refine its input feature f with respect to the database D and type information TI_D, using variable information extracted from the feature. Where the number of literals in the rule is equal to `maxliterals`, no refinement which adds a literal is considered. Having collected each refinement, the algorithm removes all those features which are not metaknowledge-respecting. Feature refinement by type specialisation is represented by the DOWNCAST algorithm. As a measure against redundancy, the algorithm will not attempt to recurse into a branch based on a feature it has already considered the refinements of.

Providing these preconditions are met, each of the constituent refinement operators are applied to the feature

and the resulting feature is tested first for validity and then for completeness. If the feature is valid, it is considered a node in the search tree. If it is invalid, then the node is rejected and the recursion stops.

Validity of a feature is tested by applying each of the following checks to it, regardless of which of the constituent refinement operators were applied.

- *Breaks cardinality.* For each method with more than one method expression in the feature, COSINUS constructs a set partition of the method expressions such that expressions in the same partition share the same input bindings. If any partition has more than the number given by the fromcard tag associated with the method, the feature is deemed to break cardinality. A similar test is done on the output bindings and the tocard tag.

- *Breaks distinct binding restrictions.* For each literal in the body, COSINUS checks that no term is repeated in any of the arguments of the literal.

- *Duplicate literals.* The literals are checked to ensure that there are no two method expressions in the feature body that share the same argument bindings and method name.

We have previously discussed the process of converting the internal form of a feature to a CORLOG— or, more properly, FLORA-2 — feature. The principal search criterion is a coverage check — the process of determining how many of (a subset of) the examples are covered by the feature. An individual O of class C is convered by a feature (body) with an interface C' for head variable O' if C is a subclass of, or equal to C', if the feature evaluates to true when O is unified with O'. COSINUS identifies a subset of the individuals during each iteration and marks them as *seeds*. When testing coverage, an individual query is constructed which tests whether there is at least one solution in the body variables where the head variable is unified with each seed example. One query is therefore constructed for each individual, with a cut appearing at the end of the query. Accordingly, the query succeeds at the first resolved solution, and the coverage count for the set of seeds is increased by one. The total number of seed examples covered by the feature is determined by the number of such seed-example queries which succeed.

Assuming the feature passes the validity checks, the feature is checked for completeness. Informally, this tests whether all the variables in the feature have been consumed by an input variable. If a feature is complete, it is adopted as a feature in the final feature set. If it is incomplete, it is further refined. If the setting `refinecomplete` is true, the a complete feature may be further refined.

A constant feature is one that is always true or false for the sample. For a large enough sample, this may suggest that the feature is trivially true or false for any example. If the setting `removeconstantfeatures` is true, COSINUS will disregard a complete feature which is constant and not include it as a candidate feature.

Definition 6.2 (feature completeness). A feature is said to be *complete* if, for each variable V in the feature, V is assigned a class and, wherever V is bound to an input variable, its values are bound by an output variable occuring previously in the clause.

Additionally, a subset of the classes can be defined to be *consuming*. This means that where they constrain a variable, the variable is already considered consumed. After a feature is adopted for inclusion in the rule set, refine recurses and the history, trace and depth count is updated in the metadata. More specifically, four versions of the refine predicate can be called, corresponding to the four broad constituent forms of the refinement

operator, namely ordered substitution, unordered substitution, adding and waking and type specialisation. We consider each of these in turn, but before doing so, we recall some central concepts.

Many of these forms of refinement rely on the availability of variables. We identified in chapter 4 that in order to reduce the redundancy of the refinement operator, only fresh variables are to be used. A list of fresh variables is kept in the metadata. Observe that fresh variables are a subset of *available variables*, those variables which have not yet been consumed in the (partially-constructed) feature.

6.3.3 Implementing substitution

Given a fresh variable, we may refine the clause by either substituting it with another variable occurring previously in the clause, or with a constant.

The constant case was considered above briefly. Where a variable is fresh, and constrained to a class conjunction in which one of the classes is subprimitive, the variable may be substituted for an instance of the class — a member of the list of values found for the class. Where not all of the members of the conjunction are subprimitive, however, it is necessary to construction the type of all values which are members of *each class in the conjunction*. In any case, a substitution is made for each member of the resulting list of values, and the variable is removed from the fresh variables list.

In the variable case, the learner attempts to find a previous variable whose class is compatible with that of the fresh variable. Variables in CORLOG features are constrained by conjunctions of classes. Accordingly, in order to perform a variable substitution, it is necessary to determine the most general common subclass of the two variables. To do this, COSINUS appends the two class conjunctions, and removes any classes which are superclasses of another. The resulting conjunction is denoted c. The user sets the maximum length of a class conjunction with the setting `maxcc`. Where $|c| \leq maxcc$, the classes are said to be compatible where c is a valid combination (*i.e.*, it covers at least one individual, does not include an abstract class, does not contain a pair which is mutually disjoint, and does not contain the superclass of another class).

Where the resulting length is greater than *maxcc*, COSINUS attempts to collapse the set of classes by identifying pairs of classes which have a common subclass which is non-abstract and has member objects, aiming to replace each pair by this most general common subclass. The original heuristic used for deciding which pair of classes from a conjunction $C = (c_a, c_b)$ to collapse into a single most common general subclass class $C' = mgcs(\{c_a, c_b\})$ is to take the pair with the highest value of s such that $s = \frac{size(mgcs(C))}{size(C')}$, where $size(c)$ repesents the number of objects belonging to a conjuctive class c. In practice, determining the number of objects in a given class can be computationally expensive, instead we simplify the check by considering those pairs of classes which have at least one object belonging to the most general common sublcass. The merging process continues either until no pairs can be found, or a collapsed combination of the required length is found. Provided this is a valid combination of classes, the substitution succeeds.

The metadata is adapted to remove the substituted variable and record the new (necessarily stricter) constraint on the substituting variable. The odc tuple is updated by adding the values for N, Ns and Np togther. The maximum order of the tuple is reset (to -1) and it may be downcasted in any position.

6.3.4 Implementing ordered substitution

Ordered substitution relies on the metaknowledge oym/4 discussed earlier, which identifies a method as being order-yielding based on its output value. We introduced the metaknowledge declaration

$$\mathsf{oym}(\mathsf{integer}, \mathsf{lessthan}(\mathsf{integer}), \mathsf{true}, \mathsf{bool}) \tag{6.2}$$

which describes an order-yielding method arising from the lessthan method on two integers (one as the host object and one as an input) when where the method returns a boolean value true. Orders may be defined explicitly via the partialorder and wellorder metaknowledge. Ordered substitution then progresses in much the same way as described in chapter 4. When attempting to refine the current feature with refine, COSINUS compares each of the literals to check whether they are order-yielding. If they are, it adapts the input argument to represent a stricter condition according to the ordering.

The principal motivation for using ordered methods is to capture the notion of a value constraint on a given variable in the system defined by the values of the input arguments. Constraints necessarily compare two values, the host object and some input term. This term can be a constant, for example in the constraint $N > 14$, or a variable, as in $N > X$. Such a variable can be unified with a previous variable in the feature or may not appear elsewhere. This contrasts with the conventional approach to feature construction in which an input argument necessarily is unified with a previously-introduced output variable at the literal addition stage. During literal addition, order-yielding methods are given new variables in their inputs which do not yet appear in the feature. In a refinement of the method expression, the variable replaced by the constant representing the least strict constant. If it is a constant, it is replaced with the next constant in the supplied order, namely, the minimal specialisation of the constraint represented by the literal. In order to reduce redundancy in the refinement process, a refinement of an order-yielding method is only performed on a method taking a fresh variable as its host object. The variable remains fresh for future ordered substitutions.

6.3.5 Implementing literal addition

A feature can be refined — made more specific — by adding a literal, or method expression, to its body. Recall that a number of measures need to be taken in order to avoid redundancy in a the literal addition operation. Firstly, a new method can only be applied to a fresh variable. Secondly, each method is assigned a number in order to induce an ordering over all the methods being used for learning. Thirdly, another form of redundancy comes about as a result of the fact that a variable can be type-specialised and, since methods propagate down the inheritance hierarchy, a literal could be added both before and after the specialisation of the variable. We therefore apply a method to a variable — *i.e.*, use the variable as the host object of the method — at the earliest possible opportunity. This implementation of addition described in this section incorporates these redundancy measures.

In short, adding a literal means identifying an appropriate binding for previously-occurring variables to classed method arguments. Finding a method expression is done in two stages. Firstly, COSINUS chooses one of the fresh variables as the host object for the new method expression, and attempts to find a set of *candidate methods* which may be applied to it given its class constraints. Since methods may be inherited down the class hierarchy, a method is deemed compatible if it is the superclass of any of the classes in the conjunction constraining the host object. Secondly, having found a binding for the host object, COSINUS attempts to find

bindings for the remaining arguments of the method. The two-stage method narrows down potentially large sets of possible methods into a small number in which the relatively complete process of constructing binding can be carried out.

Recall that each method is associated with an index, which is assumed to be positive. An addition of a literal to a existing set of literals is characterised either as an *add* (in order) or a *wake* (out of order). If we consider the maximum order (the order of the latest method in the ordering) of the set of existing literals to be M ($M = -1$ if no literals exist yet), then the new literal is added if its order m is such that $m \geq M$, and woken if its order m is such that $m < M$.

The presence of parametric classes, and in particular, method definitions which involve variables, presents a complication. For example, we may want to define a method getfirst on objects of class list(P) which return an object of class P. In applying this method, it is of interest that we determine the ground class name for P before calculating the binding. If we are to apply the method to an target object of class list(integer), we wish to bind P to the most specific class possible, namely integer, so that the class returns the most detailed type information possible. We define a *most specific parameterisation* of a class C given a class C_A of a method argument as either C or the *most specific* superclass of C which is a superclass of the class C_A given in the method.

Once a candidate method has been identified, adding a method expression to the feature based on the candidate method is done by considering each argument of the method (including its host object) in sequence. The three kinds of argument (host object, input, output) are each treated separately. The algorithm returns all assignments of terms to arguments, defining the new literal to be added. We consider the conditions necessary for a variable V constrained to a class combination C_V to match a method argument A taking a class C_A in each case.

- In the case where the method argument A is a *host object*, one of two specific conditions must apply regarding C_V and C_A. These relate to the earliest possible opportunity at which the method can be applied to V. We assume further that in the case of a parametric class, A is a most specific parameterisation with respect to the method and to V_C. Then, either (i) V is a newly introduced variable, which has not been downcasted, and one of whose classes is a subclass of, or equal to C_V, and has not yet had its type made more specific; (ii) V is an existing variable whose constraint has recently been changed via a type specification operation from C'_V to C_V. The result of this type specification has been such that C_V contains a class equal to, or a subclass of, A, but this does not hold for C'_V. Equivalently, the `odc/7` tuple takes a final element of the form $\texttt{down}(c_1, c_2)$ such that c_2 is a subclass of, or equal to, C_A but c_1 is not.

- Where A is an *input argument*, a previously-introduced variable V of class C_V is substituted, such that an element of C_V is a subclass of, or the same class as, C_A. If the method is order-yielding, an additional binding for the argument is permitted in which a new variable V of class C_A is introduced.

- Where A is an output argument, a new variable V is introduced, such that $C_V = C_A$. Order-yielding methods may additionally directly introduce a constant for the output argument.

We assume that the method has been matched for each argument in this way. If the method is an candidate to 'add' (as opposed to 'wake'), the feature is refined by simply appending the method with the appropriate bindings to the body, and resetting the output variable as the only fresh variable. If it is a candidate to 'wake', it too is appended, but it must first be verified that the resulting literal could not be added at any earlier opportunity. By this we mean that for no (partial) feature resulting from a previous refinement would this addition be valid.

For this reason, CONSTRUCTFEATURES maintains a history of features produced at previous points and their associated metadata. Each point is checked to ensure that it was not possible to apply the same refinement previously. Variables are marked as consumed in the metadata, and the greatest method, history, trails, and so on are updated. The resulting rule is then checked for validity and completeness as described above.

6.3.6 Implementing type specialisation

A further means of refining a CORLOG feature is to specify one of the constraints of a variable. Since we consider combination classes as constraints in COSINUS, this amounts to adding a new class to the combination or specialising one of the classes in the combination. The algorithm for downcasting is optimal but relies on the `odc/7` tuple in the metadata, associating six key values to each variable. The metadata follows the specification of the type specification technique presented in section 4.3.1. `odc(V,N,Ns,Np,A,P,LA)` asserts that variable V has been downcasted N times, Ns of which are simple class downcasts, Np of which are parametric class downcasts, A is the maximum order in the conjunction *at the last addition operation*, P is a list of positions at which a valid downcast can be made, and LA is the last action applied to the conjunction. P takes the values add or `down`(c_1, c_2) for adding and downcasting (from class c_1 to c_2) operations, `subst` for a tuple resulting from a variable substitution, and `create` if the conjunction has been newly-created.

COSINUS constructs a refinement for each valid class specialisation possible under the rules for downcasting a conjunction of classes. Before describing the process of finding such a refinement, we identify the main tests performed by the algorithm.

- A class combination is a *valid combination* if it is one which (i) contains at least one individual; (ii) contains no abstract classes; (iii) does not contain any pair of classes which are defined mutually-disjoint and (iv) does not contain any superclass of another class in the combination.
- A class combination is a *qualifying combination* if it is one which (i) does not contain any superclass of another class in the combination; (ii) does not contain a subclass of another class in the combination; (iii) does not contain any pair of classes which are defined mutually-disjoint.
- A class may have more than one superclass. For the purposes of the downcasting operation, if a superclass is abstract, the most specific superclass of that abstract superclass is taken as the parent. If a class has no concrete superclasses, the abstract classes are used. The first superclass in the ordering is defined as the *first parent*. The first parent relationship induces a spanning tree over the class hierarchy.
- Usually, the members of a class conjunction will be in order with respect to the ordering induced by the preprocessed class list. Where a downcast causes a class in a position to be out of order, this position is called the *order-breaking position*.

With these concepts defined, we can define the sequence of tests necessary to downcast a variable's constraint according to the rules of the optimal downcasting refinement operator. We consider the downcasting and addition refinement operators in turn. The type specialisation algorithm is called TYPESPEC, and is presented in algorithm 6.4.

The downcasting operator specifies an existing type in a conjunction. It first extracts a fresh variable from the metadata and its associated `odc/7` tuple. Next, it finds a valid position, defined by P and substitutes it for the class' minimal specialisation along the links described by the first parent relation such that the new class

Algorithm 6.4: TYPESPEC: optimal type specification

input : $\langle C, N, N_s, N_p, O, P, LA \rangle$, a 7-tuple consisting of: C, a class conjunction; N, N_s, N_p, the number of times TYPESPEC has been applied in total, to single classes, and to parametric classes, respectively; O, the maximum class in the order at the last addition operation; P, a set of legal positions for substitutions; LA, the type of downcast last applied to the conjunction; (Values for new variables are $N = 0, N_s = 0, N_p = 0, O = -1, P = \{1\}, LA = create$)

output: A set of 7-tuples $\langle C', N', N'_s, N'_p, O', P', LA' \rangle$, outputs corresponding to the above, each a type-specialisation of C.

```
R ← ∅
M ← maximum class in C
if |C| ≤ maxcc then
    foreach class c' such that c' > M do
        C' ← C appended with c'; c'_p ← first parent of c'
        if C' is a valid combination and c' qualifies and c'_p does not then
            P' ← {1,...,|C'|}; O' ← maximum class in C'
            if c' is simple then
                if N + 1 ≤ maxdc and N_s + 1 ≤ maxdcs then
                    R = R ∪ ⟨C', N + 1, N_s + 1, N_p, O', P', add⟩
            else
                if N + 1 ≤ maxdc and N_p + 1 ≤ maxdcp then
                    R = R ∪ ⟨C', N + 1, N_s, N_p + 1, O', P', add⟩
foreach p ∈ P do
    c ← class at position p in C
    foreach c' such that c' is a most general concrete subclass of c do
        if c is a first parent of c' and c' > M then
            C' is C with c' replaced for c at position p
            if C' is a valid combination then
                if C''s classes are in order then
                    P' ← {1,...,|C'|}
                else
                    P' ← {p} where p is the first position at which c_p < c_{p-1}
                if c' is simple then
                    if N + 1 ≤ maxdc and N_s + 1 ≤ maxdcs then
                        R ← R ∪ ⟨C', N + 1, N_s + 1, N_p, O, P', down(c, c')⟩
                else
                    if N + 1 ≤ maxdc and N_p + 1 ≤ maxdcp then
                        R ← R ∪ ⟨C', N + 1, N_s, N_p + 1, O, P', down(c, c')⟩
return R
```

is later in the class ordering than any class in the old combination. The class c is replaced with a new class c' in the combination. The resulting combination is checked for validity and bounds. For bounds, the algorithm checks the new values of N, Ns and Np resulting from the potential down cast, representing the number of applications of the TYPESPEC algorithm in total, and for simple and parameteric classes respectively, checking that $N \leq maxdc$, $Ns \leq maxdcs$ and $Np \leq maxdcp$. If these bounds are passed, the `odc/7` tuple is updated with appropriate values for N, Ns and Np with the last action LA set to `down(c, c')`. The value A *remains unchanged*.

Parameter	*Description*
`maxapplications`	The depth bound on the search through the hypothesis space; the maximum number of applications of the refinement operator.
`largetype`	The maximum number of objects belonging to a class such that the refinement operator will substitute variables of that class for constants to avoid production of a large number of features.
`refinecomplete`	If `refinecomplete` is true, a complete feature will continue to be refined.
`reusevars`	When a variable is consumed, controls whether it may not be reused later in a subsitution.
`mincoverage`	Specifies the minimum number of examples which a feature must cover in order for it not to be pruned during search.
`maxliterals`	Bounds the maximum number of literals (method expressions) in a clause.
`maxcc`	The maximum number of classes which may appear in a class conjunction.
`maxdc`, *etc.*	The maximum number of type specialisations per variable. `maxdcs` and `maxdcp` control the same for simple and parametric classes only.
`oymrefinement`	Allows the user to select whether order-yielding methods are refined according to the order or treated as any other method.

Table 6.1: Main feature parameters in COSINUS

The new class combination is checked for an order-breaking position. Where it exists, P adopts this position. Where is does not, P retains its original value.

The addition operator adds a new class to the conjunction. It first extracts the `odc/7` tuple from the metadata. Additionally, since the operator is going to add a new class to the conjunction, it checks the length of the conjunction against the user-specified parameter `maxcc`. If these tests succeed, it iterates through the list of classes, appending each one which is later in the ordering than any in the conjunction (according to A) in turn to the original class list, and performing the following tests for a valid choice. Firstly, it checks that it qualifies but its immediate (first) parent does not. Secondly, it must be a valid combination. Thirdly, it checks the new values of N, Ns and Np to ensure they are within limits, as described above. If these tests succeed, the combination is accepted as a new refinement and the `odc/7` tuple is updated as above, with new counts, but with A set to the order of the latest class in the class-ordering and P set to be any position in the conjunction. LA is set to add. Both the add- and downcast-operations therefore return a new class conjunction, which is possibly further refined by another call to the `refine` predicate.

We have so far considered the feature level of the search. Table 6.1 summarises the main parameters. We now consider the rule and theory levels of induction.

6.4 Implementing rule and theory generation

Having discussed how the feature set is generated, we now consider the operation of COSINUS on the rule level. Specifically, we consider how the constructed features are used by COSINUS to construct rules.

Theories are generated by COSINUS by repeatedly applying a series of steps which construct a rule. When each rule is generated, the examples which it covers are removed from the example set and a new rule is generated. The result is an ordered rule list, in which the first rule in the theory which fires is used to predict the class of the individual. The rule geration process consists of a number of distinct steps; *propositionalisation*, in which the data is converted to an attribute-value representation; *feature reduction*, in which logically redundant features are removed from the propositionalised dataset; *rule induction*, in which a propositional learner induces

a rule, or set of rules, on the propositionalised, reduced data; *back-translation and evaluation*, in which the output from the propositional learner is translated back into CORLOG— essentially FLORA-2 — rules for incorporation into the theory; *example removal*, in which the examples covered by the new rule are removed. In this section, we describe each of these stages in more detail.

6.4.1 Propositionalisation and feature reduction

The adoption of propositionalisation means that the task of rule selection may be deferred to an existing, external rule learner. This can be any rule learner which accepts a propositional or single-table representation consisting of a set of named features each of which can be true or false.

The propositionalisation step prepares the data for such a learner. We assume that prior to this step, COSINUS has chosen seed examples and constructed a feature set during its feature search. Given this feature set, it first removes any duplicate features which may have been produced as a result of redundancy in the feature generation process. A set of features are duplicates of each other if they are equal up to reordering of the literals in the *transformed* (*i.e.*, FLORA-2) rule. COSINUS then constructs a set of attributes by tagging each feature with a unique feature identifier f_i, corresponding to a feature generated in the feature search step. It constructs a data set by evaluating each feature against each example. The class of each example is assumed to be given by a target method of the form $c_i[\mathsf{class} \Rightarrow c_o]$, for a class of individuals c_i (given by the individual/2 metaknowledge) and an *object-oriented* class c_i of possible predicted classes c_o. Each feature/example combination is assigned a symbol resulting from its truth value. We adopt `t` for true and `f` for false. Each example consists of this truth value for each feature in turn combined with the class label of the individual. COSINUS does this simply by iterating through each feature, assigning an identifier to it, then iterating through each example, evaluating it against the knowledge base and assigning a symbol to it, finally adding the class of individual. By nature, propositionalisation requires the evaluation of many queries and can be one of the slowest phases in running COSINUS.

Before supplying this propositionalised dataset to an attribute/value learner, we first filter it to remove logically redundant features, as described in chapter 5 with the REFER algorithm. REFER is implemented as a Perl script, implementing closely algorithms 5.2 (top-level) and 5.1 (REFER-R). Feature ranking may be optionally performed as described in algorithm 5.3 (RANKFEATURES). Recall that REFER relies on the identification of a random starting example. Accordingly, subsequent runs may yield differing results. COSINUS optionally may be set to attempt more than one iteration and select the best run by way of the user-supplied parameter `referits`. As well as allowing us to potentially find a more reduced set of features, multiple iterations allow us to determine the variance in performance over subsequent runs of REFER. REFER returns statistics describing the number of features and neighbourhoods in each feature set produced from subsequent runs of the algorithm and the best reduction among these, both with and without feature ranking.

At this stage we have a propositionalised dataset which has been filtered to remove features which are logically redundant under some neighbourhood decomposition of REFER.

6.4.2 Rule generation and selection

The external rule learner processes the examples and produces a set of rules, usually intended as a self-contained theory for classifying unseen examples. COSINUS can use this output in one of two ways, depending on the mode undertaken by the learner. Where the parameter `mode` is set to be `single` (short for single-iteration

mode), the entire theory induced by the attribute/value learner is translated back and adopted as the CORLOG theory. Where the parameter `mode` is set to be `iterative`, the best rule is selected in the SELECTRULE step. Often, a quality ordering is assumed on the rules produced by the attribute/value learner, and the first, and necessarily best, rule returned by it is used in the covering-loop approach of COSINUS. In other cases, COSINUS may use its own heuristic to select a rule from a candidate set returned from the propositional learner. A typical objective might be to remove as many examples as possible in each iteration, and so the rule covering the most number of examples may be preferred. In general, however, it is preferable to defer rule selection to quality criteria used in the rule learner, since such heuristics are likely to be more sophisticated and more suited to the search through the rule space that the external learner performs.

COSINUS adopts a variant of the CN2 learner [19, 18] as its external learner. The variant produces an ordered rule set, also known as a decision list. Decision list learners [119] define an ordering over the rules such that the first rule in the ordering which fires is used to predict the class of the individual. Unless the rules are constructed in such a way as to guarantee that only one rule fires for a given example — as would be the case for divide-and-conquer algorithms such as decision tree learning — a resolution method must be applied to the rules to determine which is to fire. The rule firing with the largest support with respect to the training set is a typical strategy. Furthermore, the variant uses weighted relative accuracy [82, 137] measure as a search heuristic. We adopt the first rule in the ordered list at each iteration. Where the first rule is a default rule, *i.e.* one without a body, iteration ends. The generality of the propositionalisation approach allows arbitrary rule learners to be incorporated, however. In the implementation of COSINUS considered here, we simply select the first rule in the CN2-produced decision list for the iterative mode, and the full theory in the single-iteration mode.

At this stage, we have a rule produced by the propositional rule learner. This rule is necessarily expressed in terms of the feature identifiers f_i previously associated with each feature in the feature set. We now need to translate this rule back to an F-logic feature in order to evaluate it and combine it with a partially-constructed theory.

6.4.3 Back-translation and coverage testing

Given that each feature identifier f_i has been associated with a feature expressed in FLORA-2, the translation of a rule from the rule learner is fairly straightforward. We assume that each rule is a conjunction of possibly negated features and is associated with a single symbol representing the class that it predicts.

The predicted class is translated from a symbol c to a rule head `O[class -> c]` introducing a variable `O`, of the class of individual specified in `individual/2`, which necessarily form the interface for the feature set resulting from the feature search. COSINUS iterates through each of the features making up the conjunction and replaces it with the set of literals making up the corresponding FLORA-2 feature, including the class membership constraints and any inequalities arising from the adoption of object identity. All variables appearing in these literals, apart from the head variable, are renamed such that they are distinct from any variable name appearing in earlier-translated features. If the feature appears negated in the conjunction, COSINUS negates the whole set of associated literals by enclosing them in brackets and prefixing with the not operator (`\+`).

Having translated this rule, it remains to test for which individuals the rule is true and remove them from the example set. To do this, flood uses a modified version of the coverage calculation technique described above over the set of examples of the predicted class. These are then asserted as being marked and are not involved in

the example covering step at the start of the new iteration.

6.4.4 Theory construction and evaluation

At this stage the covering loop necessarily produces a set of ordered rules of its own as a theory. The decision-list-structured theories constructed by CoSinus are therefore such that the first rule in the decision list — the one from the earliest iteration — is chosen as the one to predict an example. The orderedness of the rules is ensured by appending a cut symbol to the end of each rule, thereby ensuring that where a rule makes a prediction, no further rules are considered. This orderedness holds whether the single-iteration or the multiple-iteration approach is taken, although one of the benefits of the single-iteration approach is its ability to represent a wider class of theories. For example, were CN2 to output unordered rules, they could be translated according to the same method described here.

However, the adoption of the iterative covering approach with respect to a small set E in each iteration has several advantages over performing a single-iteration approach. Firstly, fewer feature/example coverage checks are necessary. We assume that the quality function Q involves the calculation of the entailment of each example by a candidate feature. Each example is removed during one iteration only, either because it is chosen as a seed example, or removed as a result of the rule covering it. Accordingly, not all examples need to be considered, and those that are need only be considered once. Secondly, a more varied and deeper search of the feature space is made possible. The search of the feature space may vary from iteration to iteration depending on the seed example(s). Furthermore, as there are fewer examples against which coverage functions test, more of the branches of the refinement tree are pruned, and a generated feature set of the same size as in a single-iteration approach may contain features found at much deeper levels of the refinement graph. A search of the hypothesis space is effectively broken up into a series of subsearches, guided by subsets of the example set.

We conclude our discussion of the implementation by briefly considering the experimental analysis of the resulting theory. This is covered in more depth in chapter 7, so here we restrict discussion to the experimenter module of the CoSinus system, which collects statistics for evaluating the predictive power of a produced theory.

The back-translation of CoSinus produces self-contained theories in a file `theory.flr` which may be loaded and used with the Flora-2 database to predict unseen examples. The preprocessing stage divides the examples into a test set and a training set. Having performed theory induction on the training set, it iterates through each example in the test set in order to predict a class for each example. In doing so, it builds a confusion matrix, counting the number of predictions of positive and negative class labels for positive and negative examples in the data. Furthermore, the theory is constructed in such a way that when a rule fires, it returns a rule identifier (from 1 to the number of rules induced) as well as a prediction. This allows us to collect covering statistics for each rule. CN2 outputs these statistics on the training data, and CoSinus collects them on the test data. For each rule we therefore have its coverage on the positive and negative examples in both the test and training data. These are returned by the learners as two pairs of *coverage lists* in the file `theory.flr` as additional facts. `clist_training`(L_n, L_p) contains two lists such that the nth element of L_n (resp. L_p) contains the number of negative (resp. positive) examples in the training set covered by the nth rule in the theory. `clist_test`(L_n, L_p) gives a corresponding list for the test set. The use of these lists is central to the approach taken in applying ROC analysis in order to determine the predictive power of the models which is discussed in chapter 7.

Parameter	*Level*	*Description*
`referits`	rule	The number of times the feature reducer REFER is run in order to find the smallest set of features.
`seedexamples`	rule	The number of positive examples chosen as seed examples for for rule induction.
`mode`	theory	Where `iterative`, employs the covering algorithm approach. Where `single`, one iteration takes place and translated into a resulting theory.

Table 6.2: Main rule and theory parameters in COSINUS

This concludes the discussion of the rule and theory levels of induction. Table 6.2 summarises the main parameters at these levels of induction.

6.5 Conclusion

This chapter reported on the implementational aspects of the COSINUS learner, building on the feature refinement and feature set reduction techniques presented earlier in this thesis in chapters 4 and 5 respectively. We first examined the three-tiered approach to induction, in which theories consist of rules which are in turn conjunctions of features. The scheme for representing a feature was presented, and we considered how the additional data involved in this representation of a feature is used in assessing metaknowledge validity and for feature construction. The algorithms used in the generation and, in particular, refinement of the feature set were discussed, as well as the evaluation of these features and their use in theory construction.

Chapter 7

Applications

In this chapter we present a real-world application of inductive logic programming to which the object model has been applied. Object-oriented data mining aims to extract knowledge expressed in terms of representational elements specific to the object framework. This thesis argues that adopting the object model for representing data for ILP leads to several benefits for the data mining process, compared to existing state-of-the-art ILP learners with equivalent data. The object model aims to overcome a natural tradeoff in ILP systems between the efficiency of methods employing syntactic form of subsumption and the informedness of systems which incorporate background knowledge and otherwise adopt semantic approaches to refinement. The method proposed does so by defining metaknowledge under a syntactic refinement process, rather than incorporating semantic mechanisms into the refinement process.

The metaknowledge is then adopted to simplify the space of features being searched as part of the ILP task. These simplifications lead to learning benefits including a reduction in the size of the search space, a reduction in the number of impossible and equivalent features being considered, and a resulting reduction in running time complexity. The benefit can be viewed in terms of the expressiveness of the features being searched; for the same level of expressiveness, fewer features need to be searched, or for a search involving the same number of features, a more expressive feature set may be considered. In short, we aim to study a tradeoff being the expressiveness of a language and the number of features searched.

The comparative nature of the analysis requires a framework for the *equivalence* of a data mining task in the object model and in the traditional model. We test the utility of the adoption of the object model in the data mining process, and more importantly, the data modelling process. The resulting input to the data miner can be considered the facts, background knowledge, metaknowledge and learning parameters. In order for the comparison to be useful, we aim to adopt as many aspects of the data model into the input to the traditional ILP method. The equivalence between the two bodies of input knowledge are defined by a specific mapping for the purposes of experimentation. We present this mapping later. As part of this comparison, we are interested in the number of features which are selected during refinement, the number which are not metaknowledge valid and the presence of equivalence between features and its nature. We also consider the usage of the elements of the object model during the search, during feature selection by the propositional learner, and their resulting appearance in the induced theories, including the properties of this theory search.

For the analysis, we primarily consider a computational linguistics domain in which we analyse the structure of English-language sentences and derive rules allowing us to identify simple semantic roles taken by words.

This chapter is structured as follows. In section 7.1 we introduce the analysis and identify its objectives. In section 7.2 we present and identify the adopted mapping framework. Section 7.3 describes the general experimental method applied. The computational linguistics task is decribed in section 7.4, defining the learning task and the dataset it adopts. We discuss the data, and the aspects of the data which make it readily applicable for object modelling in terms of the elements introduced earlier, giving examples of the data expressed in the object model. Results are presented in section 7.5 and a general conclusion in section 7.6.

7.1 Introduction and objectives

The object model aims to exploit and integrate common properties of structured data not taken into account in existing ILP systems in a manner which is both intuitive to the user and useful to the learner. By taking advantage of these properties, expressed as metaknowledge, the process of data mining is simplified.

We first introduce some notation in order to arrive at a set of concrete claims. We compare the object-oriented data mining system CoSinus with traditional ILP learners. In this thesis, we adopt the Progol system as a representative of these traditional ILP learners. Progol is similar to CoSinus in that it uses a general-to-specific search process through a refinement graph. It uses the mode declarations and a randomly-chosen example to derive the most specific clause entailing the example, guiding the search, bounding it so that no constructed clause is more specific than the bottom clause. Its search procedure is A*-like and is guided by a quality measure based on compression[1]. Before proceeding further, we note a number of differences between Progol and the properties of a typical ILP learner assumed earlier. Firstly, terms may appear in mode declarations as well as type symbols. Secondly, the most specific clause is known as the bottom clause, and literals in it are ordered according to the input/output dependencies between them, leading to literals appearing in the same order in constructed rules. Thirdly, the search can be restricted by the user, by defining integrity constraints, general Prolog clauses which all constructed clauses must satisfy and pruning rules, which, when satisified by a hypothesis, cause the pruning of the hypothesis from the search tree.

Let us denote the object oriented learner as S_O and the traditional learner as S_T. The languages associated with S_O (here, CoSinus) and S_T (here, Progol) are L_O (CorLog) and L_T (the set of valid Prolog clause bodies) respectively. Now consider that all data D relating to a domain may be categories as extensional facts GF, intensional background knowledge B and parameter settings P. We denote by $D_O = \{GF_O, B_O, P_O\}$ the data relating to the object learner — the metaknowledge being included in B_O — and by $D_T = \{GF_T, B_T, P_T\}$ the data relating to the traditional learner. To be amenable to the comparative analysis, D_O and D_T are assumed to be equivalent, *i.e.* their elements map to each other according to the mapping described below. The set of features searched — *i.e.*, the features tested against the data — by a system S given data D and from a starting clause C is denoted $F(S,C,D)$. Where the meaning of S_O, C_O and D_O are clear, we abbreviate such that $F_O = F(S_O, C_O, D_O)$. For a system S_x and data D_x, F_x is necessarily a subset of the closure of the refinement operator ρ_x of S_x from C_x, *i.e.* $F_x \subset \rho_x^*(C_x)$.

We make some claims which set out to demonstrate that, in spite of S_O producing better theories, its search characteristics are more desirable than that of S_T.

Claim 7.1 (The learner S_O searches fewer features for comparable or better predictive performance of

[1]The value $f_s = p_s - (n_s + c_s + h_s)$ is used to guide the search. p_s and n_s are the number of positive and negative examples deducible by the hypothesis s, c_s is the length of s in atoms, and h_s the number of atoms left to complete the hypothesis The best hypothesis is the one with the highest f-value, the search is pruned if $n_s = 0$ and $f_s > 0$ and search is terminated if $f_s > f_t$ for all t in the agenda.

the induced classifier). Given a set of features F_O and F_T generated from equivalent starting clauses C_O and C_T, we claim that where the area under the ROC curve of the resulting classifiers is comparable or that of S_O is better, $|F_O| < |F_T|$. Dually, where the sizes of the features generated $|F_O|$ and $|F_T|$ are comparable, the classifier resulting from $|F_O|$ has a higher AUC.

This claim assumes a tradeoff between the feature set generated and the predictive performance of the learner. Unfortunately, it is unlikely that we will be able to control the experiments such that $|F_O|$ and $|F_T|$ do not significantly differ or similarly for AUC. Accordingly, we view the claim by saying that CoSinus presents a more favourable predictive performance/feature set size tradeoff.

Claim 7.2 (S_T searches a high proportion of features which would be classed as metaknowledge-invalid under S_O or are duplicates of other features). Given the set of features F_O and F_T, as above, generated from equivalent starting clauses C_O and C_T, the set of features in F_T which are invalid (and therefore would not be generated) is more than the set of features in F_O. Additionally we consider the redundancy of the refinement operator in terms of the number of duplicate features encountered while searching the hypothesis space. We aim to quantify this proportion in the context of the size of F_T.

Finally, we aim to demonstrate the utility of the Refer algorithm by applying it to the propositionalised data generated as part of the CoSinus process.

Claim 7.3 (Refer is a viable approach to further reducing the feature sets generated with CoSinus). Given a propositionalised feature set generated by CoSinus, Refer reduces the feature set with no significant change in the AUC of the resulting ruleset.

In order to compare S_O and S_T in a principled manner, it is necessary to adopt a mapping between the facts, background knowledge, and rules involved in the data acting as input to the learner. Defining such a mapping is a non-trivial process, since it is necessary to capture both aspects of the object model, so that the learner S_T can feasibly induce theories from it, while not attempting to re-model or emulate the object induction process. We study the issues facing the definition of such a mapping.

7.2 A mapping between representations

Much of the comparative study thus relies on a mapping between the CorLog representation and a logic programming representation. In doing so, object-oriented resolution may be mimicked to a large extent. However, the definitions of bias for most ILP learners do not allow the definition of object-oriented bias in the same level of detail as in CoSinus. There are therefore two separate aspects of the mapping. Firstly, we consider the mapping for the purposes of *evaluating clauses*. Namely, we consider what is necessary and appropriate to add to a Prolog knowledge base such that a Prolog clause mimics (a subset of) the semantics of CorLog. Secondly, we consider the mapping for the purposes of *inducing clauses*, namely, how to bring the object model into bias declarations for standard ILP. We assume a moded, typed representation as the usual basis for background knowledge.

A mapping can seek either to mimic the behaviour of object-oriented resolution and induction as much as possible, or instead be an example of commonly-used and natural data representation approaches adopted in typical ILP datasets. The former mapping, which seeks to fully implement object reasoning in a conventional

logic programming framework, produces a contrived set of background knowledge which, while being more 'object oriented', is highly atypical of a system's intended mode of use. We instead attempt to attain a mapping which represents a natural definition of the data for the non-object system considered. Where aspects of the object-oriented domain description would be naturally expressed in the domain description and background knowledge of the Prolog-based learner, they are included, otherwise they are omitted. Notions of how natural or contrived a mapping is are inherently vague. We therefore define and adopt a mapping for the purposes of comparison.

It is useful to review the bias declaration scheme assumed in the Prolog-based learner — the *types and modes* bias. Each predicate is defined as a head or body predicate — it is restricted to appear either in the head or body of a rule. Each argument takes a type symbol (possibly shared by other predicate definitions), among which no subtyping relationships are defined, determining the arguments among which substitution and unification may take place in a clause. Each argument of each predicate takes a mode, prefixing the type symbol, either input (+), output (−) or constant (#). Mode declarations restrict linking of terms between arguments.

The scope of the mapping is potentially large. Finally, a means of testing features in F_T against the meta-knowledge is determined, so that the utility of the metaknowledge can be tested as in claim 7.2. In summary, for each element of the domain model, we define an appropriate representation in Prolog, and where one does not easily exist, we consider it an extension to the representation introduced by COSINUS. The former case corresponds to the object model impacting the language and search biases and the latter the expressiveness resulting from the language extensions. The following section considers each element of the domain model and arrives at a mapping, forming the basis for experimental comparison. Parameter settings between COSINUS and PROGOL are discussed later in section 7.3.2.

7.2.1 Data

We first consider the mappings adopted between facts and rules in the COSINUS database with those in the Prolog logic program used by PROGOL.

Structured terms and functors. A structured individual is represented in logic programming languages such as Prolog by way of functors. Consider the following structured term:

```
s(np(fruit,n(flies)),vp(like,np(det(a),n(banana))))        (7.1)
```

In it, functors `s`, `np`, `n`, `vp` and `det` are used to structure non-terminal nodes of a parse tree in a structured term Structured terms necessarily represent tree-structured data, including lists[2]. On the other hand, object representations typically adopt a completely flattened representation for the data; no structured terms of this type appear in the data. Instead, structure is encoded by labelling each element of the structured term — each node in the tree described by a structured term — with an identifier, and using predicates to link identifiers. Such representations have benefits; they allow two parts of a composite object to refer to the same object, enabling graph structures, as well as simplifying the semantics of the object model and enabling the association of a class to each node in the tree. To simplify the comparison, in general we assume the absence of structured terms in the facts F_O (object database) and F_T (Prolog database). With respect to background knowledge, predicates in

[2]The functor ./2 is used in Prolog to represent lists.

B_T omit the use of structured terms from arguments passed to their heads. Similarly, methods in B_O omit the use of structured terms in their interfaces.

Class and class membership. We first consider the use of class membership for evaluation — testing whether an individual belongs to a class. The membership of an object o to a class c is represented by the CORLOG fact o : c. In Prolog, we may approximate this by defining the class as a unary predicate over object identifiers, producing literals of the form c(o). Such predicates test the membership of an object o to a class c. Similarly, membership of o to a parametric class $p\langle c_1,...,c_n\rangle$ may be tested by a literal p_b(o), for some symbol b representing the sequence of classes of bound elements c_i. Literals of the form c(o) appear as facts in F_T and in the body of rules and provide a means of testing membership to a parametric class. Such membership tests are meaningful only when applied to terms associated with types representing superclasses of c. A type declaration of the form c(+c′) for each superclass c' is therefore included matching type symbols in the type declaration of the predicates.

Using a new predicate to represent membership of each class in Prolog causes the size of the resulting bottom clause to become very large, since each predicate must be included in the bottom clause. An alternative method of modelling class membership is to consider the class to which an object could belong as a property defined by one of its superclasses, using integer symbols to represent the subclasses. We may do this by defining a predicate accepting with type declaration `ctype(+c,#int)` where c corresponds to the corresponding type of the superclass. A set of integers is defined such that each integer n_i represents c_i a possible subclass of c, and the call `ctype` succeeds if the object reference passed into the first argument is a member of the corresponding subclass. Although this reduces the size of the bottom clause, it has important drawbacks. Firstly, it cannot represent the class hierarchy in any meaningful way, and the learner is unable to descend the hierarchy. Instead, a set of possible subclasses must be nominated, which may be potentially large. Secondly, it cannot take advantage of the fact that the membership test has taken place to apply a predicate (method) to the subclass. Subclassing facts in CORLOG of the form sc :: c for a subclass sc and its superclass c may be represented by an intensional rule c(X) ← sc(X) in the PROGOL data. Membership checks of the form c(o) may then be resolved according to these rules. Generality orderings may or may not take advantage of this and other mapped background knowledge. An implication-based ordering would take advantage of rules such as c′(X) ← c(X) arising from subclass statements c :: c′ in CORLOG, producing an ordering sensitive to class, whereas a subsumption-based ordering such as θ-subsumption would not. In the case of parametric types, subclassing introduces a new level of complexity, since the parameters may or may not be subclassable, dictated by the parametric class's permissible class information in the domain. These are necessarily reflected by a typically large collection of mode and type declarations, which may be implemented via a system of Prolog rules. Recall that a parametric class' operation may or may not rely on its bound elements. In this case, the parametric class only (*e.g.* `list`) may be considered a superclass of the parameterised class with its bound elements (*e.g.* `list_integer` representing $list\langle integer\rangle$).

Traversing the class hierarchy. So far we have a convenient and natural setting for testing class membership, but this framework neglects the use of the types and modes setting for domain description in S_T. In other words, it is necessary to inform the modes-and-types learner about the class hierarchy so that it may process objects which are defined at different levels in the class hierarchy. Two basic properties need to be captured. Firstly, predicates specify a type in their input. In COSINUS, in order for terms in these types to unify, conditions

between the types expressed in terms of the class hierarchy must be satisfied. In traditional ILP, including PROGOL, these types must be equal. We therefore either ignore the class hierarchy or seek to mimic the inheritance mechanisms of CORLOG. A number of approaches are possible. Firstly, we might include a new type declaration for each combination of valid input and output types. Such an approach is likely to lead to a large number of declarations, expanding PROGOL's bottom clause and possibly leading to ambiguities in the choice of the output type. Secondly, in order to propagate the application of a superclass' method to its subclass, we can transfer a term to one of a new type by introducing a rule `upc(X,X).` and associated type declarations `upc(+sc,-c)` for each assertion of the form sc :: c. Applying `upc/2`, which we informally term *upcasting*, thus introduces a new term associated with the superclass' type, allowing the application of a predicate to it as a target object or and input. However, this practice is both contrived and over-complicates clause generation, producing long clauses and introducing new variables. Furthermore, the situation arises where several variables in a rule represent the same object, but each assigned to a different type. In the newer variables, type information is lost and further refinements to the clause become more complex. Among these refinements, more features are likely to be logically equivalent but for the `upc/2` literals, which serve only to satisfy the typing requirements for the ILP learner. All of these factors together cause a dramatic increase in the size of the hypothesis space. For this reason, this necessary adaptation to the learner is not adopted in our study. Where it is necessary for a predicate to take a set of classes (the set of subclasses of its 'main class'), we introduce multiple type declarations.

Finally, one of the central mechanisms of the object framework presented is the provision of refinement via *downcasting*, in which a variable's class is restricted to a given class. No such mechanism can easily be mimicked in a natural representation of the background knowledge in Prolog, since the predicates which check class membership do not introduce any type information into the induction, or clause construction, mechanism. In other words, the restriction inherent in including a class membership literal is not reflected in the typing system. Where classes are checked with the $c(+c')$-predicates (for c' a superclass), we may choose to define additional rules of the form $c(X,X) \leftarrow c(X)$ and use instead type declarations $c(+c',-c)$, thereby introducing a new output argument (and variable) in a literal $c(X,Y)$, for which X is of class c' and Y of class c, the class to which X has been restricted. However, in practice, this is both contrived and wasteful of resource bounds which may exist on the learner. As such, progressive downcasting of the type symbol associated with a variable is difficuly or impossible in learners adopting the modes and types bias. In other words, 'refinement' is restricted to class membership checks only.

Method calls and their signatures Method calls in CORLOG have a close correspondence with Prolog goals, especially where it is assumed that the arguments are moded. We represent method signatures by the type and mode declarations in the background knowledge. In these declarations, the host object and inputs of a method call in CORLOG appear as arguments to a mapped Prolog predicates, moded as inputs. Similarly, outputs from the method call appear as arguments moded as outputs. Additionally, the classes associated with each argument appear as type symbols in these mode declarations. Moding, a purely syntactic restriction in many inductive learners, using type symbols which match or do not match, cannot take into account the class membership facts in the database nor subclassing relationships, and therefore appear only as an approximation to the class of an object-oriented argument. The mapping of calls which obey these signatures follow the same form. Class membership checks may be added to the clause to ensure passed objects are members of some class defined in the method signature, causing the failure of the method if the arguments are not of the correct type.

For example, the CORLOG method signature integer[lessthan(integer) ⇒ boolean] is mapped to the mode and type declaration `lessthan(+integer,+integer,-boolean)`, the specific call 2[lessthan(3) → true] being mapped to the logic programming literal `lessthan(2,3,true)`. Methods in the object model may be inheritable or non-inheritable. *Non-inheritable* methods may only be applied to objects of a particular class. In a Prolog database, since there is no notion of class or type hierarchy, and non-inheritability may be implemented by omitting declarations whether the multiple declarations approach is taken. Where re-typing is adopted, it is necessary to split a class' methods into two new versions of class, one permitting upcasting (for inheritable methods) and the other not. We do not adopt this practice for the purposes of our comparison, however.

7.2.2 Metaknowledge

The body of metaknowledge defined in an object model can be taken account of during the process of induction in two main ways. Firstly, we can aim to reflect metaknowledge as much as possible in the declarations and other information taken as input by PROGOL in order to allow it to guide the search. Secondly, we use metaknowledge to test whether a clause violates the metaknowledge.

Some clause-validity notions, such as linkage and decomposability, are independent of whether a clause employs an object-oriented approach. Most metaknowledge in CORLOG serves principally to constrain the language of valid clauses and the search through them. Additional background knowledge guarantees certain properties of the data. Partly because of their use of the object model, they are not typically part of ILP systems, and additionally are not usually convenient to express in Prolog. Most ILP systems cannot exploit ordering, for example. More generally, logic programming has no intrinsic concept of class and does not encourage the structuring of data according to the object decomposition. As a result, concepts such as abstract classes and disjoint and dimensional inheritance do not appear in the traditional ILP data model. Cardinality and functionality are represented by a number of ILP systems. In some ILP systems, this is done at the variable-sharing level, meaning that they may be used at most n times with the same bindings of input variables. In others, the number of times a predicate may appear in a clause with *any* argument bindings is limited, either by a specific recall number associated with a predicate. Cardinality relationships may therefore be introduced from the object model to a correponding set of arguments in the predicate declarations.

An inductive bias based on types and modes does not lend itself to incorporation of metaknowledge from the object model. Translating such object knowledge to a types and modes setting is often problematic or impossible. However, it is possible to test a clause constructed by the traditional ILP system for their violation of conditions introduced by metaknowledge. Conditions for violation of metaknowledge for PROGOL follow in a straightforward manner from the mapping defined between CORLOG features and conventional logic programming clauses. We briefly review the tests for a valid clause under metaknowledge introduced in chapter 3 in the context of checking traditional ILP clauses for metaknowledge-validity. Where appropriate, these tests assume a syntactic rewriting of the CORLOG syntax used in definitions into a corresponding syntax for conventional logic programming, under the mapping identified.

The first form of metaknowledge concerns the signature expressions of CORLOG and the valid variable sharing between the typed arguments that they introduce. Definition 3.17 introduced the concept of a signature-respecting clause. ILP learners based on modes and types do not unify terms of differing types. However, PROGOL often unifies terms of differing types if the types overlap, *i.e.* if the two types share individuals. Accordingly, clauses require analysis to check whether the variable sharing is valid under the type scheme

Element	CORLOG *representation*	*Facts and declarations*
Structured terms	flattened object facts	flattened Prolog terms
Lists[a]	list object attribute	lists remain unflattened
Simple classing assertion[b]	o : c	c(o)
Parametric classing assertion	o : c(a)	c_a(o))
Class membership test	o : c	c(o)
Mode/type declaration	—	c(+c)
Subclass assertion	c :: c′	c′(0) ← c(0)
Signatures	$C_O[m(C_A) \Rightarrow C_R]$	$m(+c_o, +c_a, -c_r)$
Methods[c]	$O[m(A) \rightarrow R]$	m(o,a,r)

[a]Lists are the only form of structured term permitted in the data.
[b]Refers to a fact which *asserts* membership of the primary class.
[c]Attributes in F-Logic and CORLOG are considered to be method calls with one host object and one output argument. This convention is continued in the mapping presented.

Table 7.1: Mapping data from CORLOG to Prolog.

adopted.

In practice, a program analyses the clause syntactically to determine whether it is valid under the object metaknowledge. For each variable, it infers type information about the variable from the position it occupies in its containing predicate. Each variable is associated with a set of classes which it adopts when considered in the context of the object representation, considering input, output and the analogue of class membership. The validity of these classes is tested with respect to the criteria given in chapter 4. Redundant class membership testing — where the class membership literal contributes nothing to the meaning of the rule — is also tested. In the same way, the presence of a test for an abstract class, two tests for a pair of mutually-disjoint classes, or a set of literals which break multiplicity bounds, are also considered metaknowledge-breaking. A feature generated in a PROGOL-based ILP system which does this is termed *invalid*.

7.3 Experimental method

In the previous section, we considered a natural correspondence between the CORLOG approach to knowledge representation and one in commonly-used Prolog with an associated modes-and-types bias. This standard correspondence is summarised in table 7.1, and gives a point of comparison with usual data models in traditional ILP, highlighting the differences in search between COSINUS and traditional ILP learners. While Prolog may be contrived as much as possible to follow object-oriented reasoning — employing the most object-oriented data model possible under Prolog — we wish to analyse the operation of ILP on a typical representation of the data, under modelling methods which reflect *typical* use.

Before considering domain-specific investigations, we establish the general experimental method adopted in our comparative evaluation of COSINUS and an established ILP learner. The method sets out to verify the claims made in section 7.1 by comparison with PROGOL. Its types-and-modes bias is comparable to COSINUS and its framework CORLOG.

In this thesis, we compare the behaviour of COSINUS and PROGOL. COSINUS uses a syntactic search method employing metaknowledge from the object model. PROGOL, the conventional ILP learner, employs a Prolog-based representation and a syntactic θ-subsumption-based refinement operator. Since it is syntactic, it cannot take advantage of intensional background knowledge and may produce clauses which are invalid,

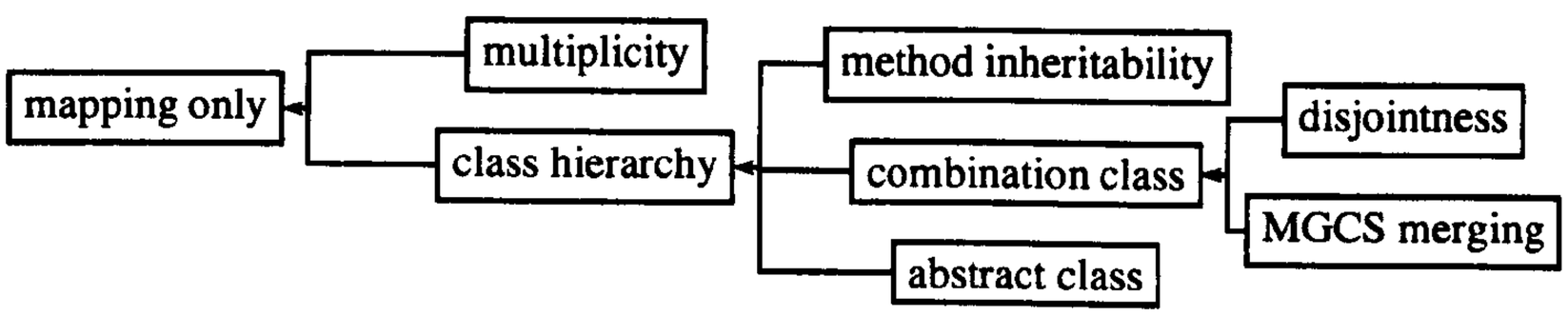


Figure 7.1: Variants of CoSinus resulting from omitting and including elements of the object model.

redundant or impossible under when mapped to CorLog.

We are interested in the utility of various elements of the additional object model during the search process itself. As such, it is of interest to consider the effects of including or omitting these aspects. For example, the presence of class membership, possibly with respect to a class conjunction, and possibly incorporating parameteric classes, is a key feature of the CorLog framework. Objects are necessarily situated within a hierarchy of classes, over which methods may propagate via inheritance. We may omit mechanisms such as this inheritance behaviour, multiplicity, specialisation of a term's class during refinement, the metaknowledge describing disjoint and abstract classes and downcasting during unification, in order to understand how these contribute to learning in the object model.

During the CoSinus refinement process, the clauses which are redundant as a result of metaknowledge declarations are not considered as valid features. These features may be disregarded, and not refined further. We illustrate this reduction in the space of features searched by considering the size of this space with and without these forms of metaknowledge. The reduction in the size of the hypothesis space comes about in several ways. Restrictions on argument bindings, in particular the restrictions on and between bindings for the host object, input arguments and output arguments, linkage and decomposability limit the set of valid clauses, as well as type-correctness. Further restrictions are placed on co-occuring classes from the subclass hierarchy, and rules on subclassing in parametric classes, and the semantics of disjoint, abstract and dimensional inheritance. Finally, redundancy and impossibility in the presence of partial and total orders, cardinality, exclusivity and inverse methods also reduce the hypothesis space. Aspects of the object model may then be divided into those aspects which are representable under the mapping. The first category are those which introduce some new level of expressiveness in the language not intrinsically part of the Prolog representation, for example the class hierarchy, including parametric classes. The second category are those which guarantee some property of the CorLog clause and mapped Progol clauses. Inheritability, cardinality, combination classes, disjointness, totality and submethods, as well as linkage and decomposition metaknowledge, may be used in this way.

As well as introducing language bias, several elements of the object model also introduces search bias. Subclassing relationships between simple classes, conjunctions and parametric classes guides refinement differently than under traditional ILP systems which treat them only as simply-typed monadic predicates. The Progol learner is used as a baseline against which variants on the CoSinus learner are compared. These variants represent the CoSinus learner with more aspects of the object model are introduced. Each variant exhibits a different search behaviour, producing a different set of searched nodes and refinement lattice. The variant comprises metaknowledge which may be used to determine the validity of clauses in the Progol run. Some aspects of the object model necessarily depend on each other. For example, in order to consider whether a method is inheritable or not, the notion of class must first be introduced. These dependencies give rise to the family of CoSinus variants possible for comparison. This family may be visualised as a tree as in figure 7.1.

7.3.1 Measurables

In performing the comparative analysis, we measure the following properties of the hypothesis space, the refinement tree spanning it, the search method, and the resulting feature set. Additionally, the nature of the search through this hypothesis space to obtain a set of features is of interest. Recall that CoSINUS comprises a two-level search process, in which clauses are found during successive iterations and a theory built up from each of these clauses. In this approach, we adopt a single propositionalisation of the data and use CN2 to construct a theory from the features. PROGOL instead seeks to construct a single classification rule. We demonstrate that CoSINUS's approach offers a large efficiency increase over PROGOL as a result.

In the PROGOL experiments, searching for a *theory* involves multiple iterations of the search through the refinement graph, at each stage considering a subset of the examples used in the previous iteration. Accordingly, we first consider important results for each iteration before considering those for the higher-level theory search. Measurables of interest at the *iteration* level on the search of CORLOG features as well equivalent Prolog clauses, are based on the *multisets* (*i.e.*, allowing repeats) of candidate queries searched, and its subset, the queries selected according to the selection criteria in the parameters. We then compare the following: the number of candidate queries searched against the number of queries found to be supported by the search heuristic, and the proportion describing the number of selected queries over all candidate queries; the set of candidate and selected features respectively, which are distinct under feature equivalence, and the size of these sets; the subset of features which are metaknowledge-valid under the mapping between CORLOG and Prolog. Furthermore, for CoSINUS we consider the number of features selected for propositionalisation.

In experiments, we compare the iteration-level measurements above averaged over each iteration as well as the total number of iterations required to construct a theory. Additionally, we are interested in comparing equivalence and repetition of features *between* iterations. The number of interations which share an equivalence class of features indicate whether new portions of the hypothesis space are being searched on subsequent iterations, or whether it is the case that subsets of the hypothesis space are being searched instead. Measurables of interest at the *theory* level are as follows: the number of covering iterations taken; the multiset union of the candidate and selected features across each iteration, and their size; the degree of duplication of features across successive iterations; and the amount of CPU time taken for induction. The experiments were run on a 2.4GHz AMD Opteron-based computer. The time recorded excludes compilation time. The area under the ROC curve applied to a test set, is used to estimate the predictive performance of the induced model.

7.3.2 Experimental parameters

In this section we summarise the parameters used in the experimental method. In chapter 6, we considered parameters guiding the learning process, which we will refer back to in this chapter. We therefore consider here only experimental parameters specific to the investigations carried out here.

Firstly, the approach taken to imitating the inheritance hierarchy — the mapping approach adopted between CoSINUS and PROGOL data — forms another kind of parameter. This can be summarised in the two following settings. In E_1, we adopt an approach to background knowledge which introduces a new class membership testing predicate for each class in the system. Each class in the object model is represented by a type in the Prolog-based system, and additional declarations may be used to determine where these types are compatible. In E_2, the principal difference is that class member is instead done by considering the class to be a property of its superclass, for example where a fact `vg(X, 1)` ← `perfectiveverbgroup(X)` holds. This gives one predicate

for each superclass, testing the membership of one of its subclasses as a property of the individual. In both cases no new variable is introduced of the subclass' type.

Additionally, we consider the parameters of the underlying propositional learner. In this thesis, we adopt the ordered CN2 learner and its variants as the propositional learner. The star size is fixed to 20, while the significance threshold and version may be set with parameters `cn2threshold` and `cn2version`. The version may be the original CN2, a version for subgroup discovery, or CN2 using weighted relative accuracy. We adopted the weighted relative accuracy version for this work, with all other settings being the default.

Variant forms of COSINUS— those that omit some aspect of the object model — are defined by another set of parameters. For example, *object identity* (`objectidentity`) causes the addition of extra literals to induced and evaluated clauses which ensure that the values of distinct variables are never equal. The remaining variants considered are as follows: *multiplicity* determines whether to omit multiplicity metaknowledge during feature construction; *class specialisation* enables or disables the refinement of a term's class during refinement; *abstract and disjoint classes* removes the class metaknowledge during learning; *method propagation* allows a method in a class' superclass to be called — disabling it means that a method may only be applied to a term of its exact class, in effect making all methods non-inheritable; *Merging in MGCS* considers the situation where the unification of two terms results in a combination class greater than the maximum permissible length — where merging is permitted, two of the classes are replaced by their most common general (non-empty) subclass in order to reduce the length of the class. The COSINUS learner was adapted to disable and enable these aspects of a variant.

For comparative purposes, the parameters chosen for PROGOL should aim to correspond to their equivalents in COSINUS as much as possible. We briefly review the main parameters in PROGOL which correspond to those of COSINUS and consider these equivalents. Firstly, the recall setting on type declarations in PROGOL relates to cardinality settings on COSINUS methods. In order to limit the search process, PROGOL takes a parameter for the number of nodes expanded in the search (setting `nodes`) and a limit on the number of atoms in a hypothesis (setting `c`). The length limit in COSINUS is on the feature-construction level rather than the rule-construction level, and so these are not comparable. However, we compare two settings of `c` in order to understand how PROGOL performs on different bounds. Finally, the number of bottom clause iterations (setting `i`) bounds the variable depth of clauses explored. Except where indicated, default settings are used.

7.4 Analysis of natural language

The use of object orientation to model grammatical structures has has a long history in computational linguistics. Grammatical structures and categories are used in grammatical rules defining grammatical objects in terms of their constituents, such as S ← NP VP, meaning 'a sentence is a noun phrase followed by a verb phrase'. These rules thus define a grammatical object in terms of its constituents. In order to model language adequately, these constituent categories can become very complex. Verb phrases are a case in point. In sentences using verbs such as WANT or NEED, a verb phrase may contain another — the verb phrase TO ARRANGE A PARTY is enclosed in I WANT TO ARRANGE A PARTY, with WANT the *complement* of TO ARRANGE A PARTY. We might want to model this behaviour with a rule. However, these verb phrases may only be included in this way with WANT if it is infinitive, and not every verb phrase may take a infinitive complement in this way. There are many such properties of categories, which must be taken into account when modelling grammatical behaviour. As a result, phrases in a grammar are often given detailed categories such as *third person singular noun phrase* or

modal verb phrase. Defining separate grammatical rules for each detailed category leads to an explosion in the number of rules necessary to capture the behaviour. To overcome this, often these categories are arranged into hierarchies, in which transitive and intransitive verb phrases are both kinds of verb phrase, for example, and rules are defined at the appropriate level. Typically, grammatical rules model dependencies between a whole and its part, and hierarchies of grammatical categories are useful. However, phrases and words in a sentence may depend on each other. Subject-verb agreement is an example of this. For example, we say THE DOG EATS HIS FOOD, but not THE DOG EAT HIS FOOD, since the form of the verb EAT must agree with whether the noun (dog) is singular or plural. If the grammar is context-free, a further dimension of grammatical categories become necessary, for example *verb agreeing with a third person singular noun phrase*, further increasing the set of possible categories in order to model defined properties. This very fine-tuned categorisation becomes necessary because so many categories possess defined properties which other grammatical categories, including their more general categories, do not possess. It is meaningless to talk about the tense of a noun phrase, or the modal verb possessed by a verb phrase which is anything other than a modal verb phrase. As such, grammatical structures in a given sentence may easily be viewed as *objects* belonging to *classes* which have complex sets of properties (methods and attributes, in the object terminology) associated with them. Adding named properties such as 'plurality' to more general grammatical categories is termed parameterisation in the linguistic literature, and may be seen as an analogue to defining method and attributes for particular object-oriented classes on an inheritance hierarchy. The ability to refer to other objects in a database also permits the linkage of grammatical categories and their constituents to others in the same sentence.

The object-based view of natural language data has been studied for some years under a different name in linguistics. *Constraint-based formalisms* in linguistics aim to solve these issues, by defining a simple constraint language on properties of a grammatical category, forming *feature structures* [58]. In a *feature structure*, instead of representing each grammatical category with a symbol, attributes are associated with grammatical categories, for example a property *number* denoting whether it is singular or plural. The same can be applied to verbs, and grammar rules can be defined as only succeeding if the property *number* matches. Each attribute is constrained to take on a number of possible defined values, and structures possess substructures, which may be shared with other structures. Unification rules are defined across these structures also; two structures may be *unified*, or merged, into one resulting structure if a valid assignment exists — where no two specified attribute values in a structure differ. Feature structures thus model partial information about a (linguistic) object, in terms of value constraints which may subsume each other and form generality orders over feature structures. The unification process performs specialisation along this order. The constraint-based formalisms and the object-oriented logic programming formalisms previously introduced closely correspond. Where a feature structure's attributes are fully specified, it represents an object, or composition of objects. Where attributes are unspecified, the feature structure represents a class. The feature structure formalism may be extended to one involving a inheritance hierarchy of types, more specific types inheriting their parents' properties. Aït-Kaci [4] adopted logic programming ideas on unification to propose unification of inheriting feature structures. Later, typing of feature structures was proposed, including the range of possible values that an attribute may take. The typing mechanism of feature structures associates each feature structure with a type, with each type possessing conditions stating which features may be included in it, ensuring each attribute is meaningful. Unification is then done according to type as well as value, with the unification of two simple types being the most general type more specific than both of them. Unification with inheritance followed with the proposal of KL-ONE. Within computational linguistics, inheriting feature structures have been used for (linguistic) knowledge

representation involving inheritance as in the implementation of dependency grammars. A feature structure therefore represents a set of constraints on a grammatical structure, or a *pattern* which can be matched to a set of grammatical structures. These constraints have a corresponding representation in first-order clausal logic, and the typing, object-identity and inheritance rules extend this correspondence to languages such as CORLOG. The set of permissible feature structures relate closely to the signature atoms and the is-a definitions in the database. The use of feature structures in linguistics demonstrates the applicability of object models to grammatical structures in linguistics, and motivates it as an application area. Feature structures have been used in many of the levels of linguistics, particularly phonology, syntax and semantics. Here we hope to demonstrate that object-orientation is a suitable representation paradigm for the induction of rules (analogous to feature structures) describing regularities in grammatical structures.

7.4.1 Applying inductive logic programming to computational linguistics

Many approaches to linguistic analysis of text employ statistical methods to determine general patterns. Other approaches describe common patterns by constructing grammars. In contrast to both of these, the propositionalisation approach adopted in the learner requires the knowledge learnt to be in the form of classification rules. Therefore, we consider a number of simple tasks in linguistics which are well-suited to formulation as a classification problem. Appropriate classification tasks may be divided into the linguistic levels of syntax, the study of the rules that govern the way the words in a sentence come together and morphology, the study of word-formation from smaller units. Firstly, we review a number of computational linguistics tasks which have already been attempted by applying ILP systems.

Mappings between words such as from the present to the past tense of English verbs is one area that has been explored. Analogical past tense prediction [102] predicts the past form of a English verb given its present tense, using an analogical approach. In another approach [37], gender and case were used to learn mappings between the lemmatised or stem form of nouns and their oblique or inflected form in Slovene. ILP was also applied to the task of determining whether the noun and verb in a sentence are closely linked by a role in a French corpus of engineering texts [20]. This link is termed the N-P relationship. For example, KNIFE and CUT are linked by a telic role, BUILD and HOUSE by an agentive role, HANDLE and CUP by a constitutive role, and CONTAIN (in the sense of containing information) and BOOK by a formal role. A pair like CORROSION and CHECK are not considered linked. The work uses ALEPH to learn rules which detect the presence of such roles in a sentence. Finally, irregular French verbs were segmented and stemmed using a combination of genetic algorithms and ILP [63].

Rules may also be induced which *label the role of a word* in a sentence, for example its part of speech. This is an interesting task because of the natural ambiguity in natural-language text. Part-of-speech labelling with ILP has been applied to datasets describing sentences in English [23], Hungarian [57], Slovenian [24, 64] and Swedish [86]. At the sentence level, the problem of PP-attachment [62] has been approached using ILP. In this task, the selection of one of two possible parse trees for sentences of the form (VERB NP PP). For example, the phrase PETER READS A BOOK ABOUT COMPUTERS, in which the prepositional phrase is adjectival, *i.e.* the phrase 'about computers' relates to the noun and, has the parse tree (VP (VERB NP(NOUN PP(PREP NOUN)))). The phrase PETER READS A BOOK ON THE BUS, in which the propositional phrase is adverbial, *i.e.* the phrase ON THE BUS relates to the verb, and has the parse tree (VP (VERB NP(NOUN) PP(PREP NOUN))).

The induction of *grammars and parsers* is another common kind of learning task used when applying ILP to computational linguistics. An approach to semantic interpretation using a quasi-logical form has been investigated [25]. Natural-language shift-reduce parsers were also induced from the WordNet semantic lexicon and the treebanks [61] — text corpora annotated with its syntactic structure. Finally, the STO system [12] induces *transfer rules*, describing mappings between phrases expressed in natural language and restricted logical representations of their meaning. The intermediate representation is known as quasi-logical form (QLF). Logic-based approaches to linguistic analysis often involve the induction of context-free grammars. Context most commonly expresses word dependency, in which a word in a sentence depends on the presence of other words, potentially not appearing in the part of the parse tree covered by a grammar rule. Such dependency may be modelled in the frameworks of finite-state machines and context-free grammars, after their formalisation by Chomsky [17].

The object domain provides several data modelling facilities relevant to the study of natural language. The class structure provides a dual purpose of enabling the definition of levels of parts of speech (word/verb, verb/transitive-verb, etc). Objects facilitate the modelling of categories, or protoypes, in which more general concepts represented by words can be defined in terms of more specific concepts (FURNITURE/CHAIR), *representative* specific concepts (compare BIRD/ROBIN to BIRD/PENGUIN), and elaboration of concepts by overriding (in which a ROBIN flies, but a PENGUIN does not). Container classes naturally model higher-order semantics of natural structures such as sequencing and parse trees in the data. Meta-knowledge regarding general rules about sentence structure, *e.g.* the assumption that a noun is described by at most two adjectives, or the knowledge that a monotransitive word takes only one grammatical object, may be incorporated. Word properties based on their part of speech, for example the role of head and functional words in subtrees of the parse tree. For example, the head of a verb phrase is its verb and a noun phrase its noun. Similar relationships exist for sentences.

7.4.2 Data and corpora

Having reviewed some of the data modelling issues with grammatical categories, we can already see that data from linguistic corpara is both rich in structure and (necessarily) incorporates a strong notion of inheritance, making it highly suited as an application area for inductive logic programming and in particular ILP using the object data model. The data has a strong notion of individual; a sentence with its associated parse tree may be considered an individual comprised of its constituent clauses, phrases and words, relating to the compositional aspect of the object model. Arbitrary structures may be captured by the flattened object data set. The large set of grammatical categories are well-represented by a class hierarchy, leading to large, connected class structures. Furthermore, within this class structure, more specific categories typically introduce more properties as they become meaningful, for example the plurality of a noun phrase. Furthermore, sentence are additionally structured as a *sequence of words*, each of which carry their own properties and substructures. Finally, meta-knowledge is useful in capturing a number of elements of the model. For example, grammar rules for English present cardinality constraints (a clause has only one subject and one verb), numerical properties of sentences such as clause count allow reasoning over ordered sets, and incompatible combinations of classes naturally give rise to disjoint sections of the class hierarchy.

For a learning task to be successful and demonstrative, we rely on a linguistic corpus which is both detailed and correct, and whose representation may be easily adapted into an object database. We briefly review suitable

corpora and data which may be used to augment them. *Treebanks* are text corpora in which each sentence is annotated by a highly detailed tree structure describing its semantic properties, and are used to train parsers, but also are applicable to a wide range of computational lingustics tasks. Many treebanks exist and have been used in the literature, among the most common the Penn Treebank and its Wall Street Journal corpus [92], which associate words in a corpus with their parts of speech using some syntactic scheme such as brackets.

The SUSANNE [124] corpus is a free and highly-detailed corpus based on the common Brown corpus, emphasising the logical and surface grammar of the English language and covering a wide number of different structures in English, at the document, sentence, clause, phrase and word level. Comparative to other corpora, the scheme is wide-ranging, aiming to unambiguously represent many aspects of English grammar. Each clause and phrase is associated with a *form* tag, categorising the type of clause or phrase (noun phrase, verb phrase, etc.) and applying sub-categories of each type to these (such as 'singular noun phrase' or 'negative verb phrase'). These are supplemented by additional modifiers specifying categories such as interrogative, imperative or subjunctive clauses, as well as marking phenomena such as subordinate clause structure. The *function* tag defines the semantic role of the phrase or word in the sentence, distinguishing between categories such as subjects (logical and surface), objects (direct, indirect, logical and surface), preopositional objects, complements and agents. Adjunct function tags define additional forms of semantic role in the phrase, for example phrases describing places, direction, time, manner as well as more abstract concepts such as contingency, respect and aspects of particular phrase types such as participles and relative clauses. Finally, additional links between nodes on the parse tree are provided by numbered indices, which link together distanced nodes in a parse tree, such as those links introduced by a relative clause.

7.4.3 Data preparation and preprocessing

The SUSANNE corpus was selected for the computational linguisitics experiments in this chapter. Thirty-three documents from SUSANNE were adopted[3]. The SUSANNE consists of a series of plain text files, each of which contain a line for each word in the corpus, describing its part-of-speech, lemma, and a segment of the parse tree. This data was preprocessed by a program called MAKETREE, which processes these elements of the data to a form suitable for reading into ILP systems. Two formats were produced. The first consists of a set of F-Logic (and therefore CORLOG) facts. The second consists of the Prolog equivalent under the experimental mapping described above. The data — facts or 'libraries' of intensional rules — in the resulting logic program may be divided into a number of subsets which cover different aspects of the corpus' structure. We discuss each of these categories in turn, briefly discussing the preprocessing undertaken by MAKETREE.

For brevity, we summarise a number of subsets of the data not used in the experiments in this thesis. *Lemmatisation* considers the stem, or lemmatised form, of words in the corpus. The lemma of SAID is SAY, for example. Features of the lemma form may then be associated with the data. Each word also carries WordNet annotations [95], describing the exact sense of a word by linking it to an entry in the WordNet database, a semantic lexicon of English linking words together as hypo- and hypernyms, holo- and melonyms (in nouns), and many other high-level relationships. *Indices* link two arbitrary nodes in a parse tree, denoting referential identity between them, bringing a new structure additional to that of the parse tree, namely referential structure. An example from Sampson is the sentence JOHN EXPECTED MARY TO ADMIT IT. Here, the clause TO ADMIT

[3]Specifically, documents A01, A02, A11, A12, A13, A14, G01, G11, G12, G17, G18, G22, J01, J02, J03, J04, J05, J06, J07, J08, J09, J10, J12, J17, J22, J23, N05, N09, N10, N11, N12, N14 and N15, which contain additional information on synonyms not exploited in this study.

IT is linked to MARY. Finally, *sentence-level* data associates a unique identifier with each sentence and reports the text of the sentence.

Part-of-speech annotation and wordtags. Part-of-speech data is included for each word in the corpus. Each word is categorised in a very detailed, hierarchical categorisation scheme consisting of approximately 350 categories represented by symbols and listed in [124]. We adopt the Penn treebank, a simpler and more commonly-used scheme for the representation of part-of-speech annotation. We use a mapping between the SUSANNE corpus to translate to one of approximately forty specific categories, each of which belong to one of nine general categories. This category structure is represented by a class structure, allowing part-of-speech annotation at two levels of detail.

Parse tree. The parse tree accompanying each paragraph is the main form of structure in the SUSANNE corpus, describing the grammatical and typographical elements of the documents. Such a tree has grammatical categories at its branches and words at the leaf and is represented in both knowledge bases as a flattened structure of object identifiers, linked by parent/child relationships. The *node* thus becomes the primary unit of structure in the tree, and refers to a grammatical category adopted by one or more words. Each node is thus identified directly or indirectly with a set of words. MAKETREE introduces a new object for each node, describing parent and child relationships between them. As nodes are ordered under their parents, each node has a number associating its position in the sequence number among its siblings. This is necessary due to the fact that we do not consider lists in our data. A containing node clause or phrase may be found by means of the predicate `containscp`. The 'scope' of a node — its parent and siblings — may also be found. Finally, the notion of 'treewise next' and 'treewise previous' nodes are defined in the domain library, using the tree structure to define successive and previous nodes in the tree.

Definition 7.4 (treewise next, treewise previous). A node N' is treewise next (resp. previous) to a node N if (i) N and N' share a common parent P, such that N' is linked through the next (resp. previous) child of P from N and (ii) N' is linked to P by first (resp. last) child links.

Form and function tags. Form tags label the nodes of a parse tree and provide information regarding the internal properties of the words dominated by the node, and usually determine the grammatical properties of the words, such as the category of clause or phrase they represent, rather than their role in the sentence. Four levels of structure are included — rootrank (paragraphs, headings, quotations and interpolations), clausetags (sentences and their subclauses), phrasetags (noun and verb phrases, *etc.*) and wordtags (parts of speech). Each level is arranged into a hierarchy of grammatical categories which may be augmented with additional adjunct properties. Each node, represented by an object, is assigned a class, and the subclassing hierarchy models this hierarchy. Function tags identify semantic *roles*, and serve to label nodes as subject or otherwise, the basis of our classification problem. *Complement* tags occur at most once in a clause and break it down into a set of roles, for example the subject and object of a sentence. *Adjunct* tags may occur many times and elaborate on the role, stating whether the role is a place, direction, time, and so on. Like form tags, MAKETREE introduces classes to represent the association of these form tags with a node.

Dependency and head nodes The SUSANNE corpus is unusual in that it does not take into account dependency structure in the parse tree, but instead takes a phrase-structure basis to English grammar. The PRINCIPAR

[85] system aimed to automatically augment the SUSANNE corpus with dependency information. The dependency domain library aims to augment the structure of SUSANNE to include such information about the head of any arbitrary grammatical structure — *i.e.* node — in the knowledge base. Such dependency information is very important in the analysis of English documents and introduces a *dual view* of the parse tree data. The parse tree as translated by the SUSANNE corpus by MAKETREE consists mainly of parent, child and child-sequence relationships. The dependency tree consists of the same nodes but defines different relationships among those nodes. The methods *dparent*, *dchild* and *dseq*, implemented as intensional knowledge, implement these relationships. *dparent* is a simplified form of dependency, adopted both for computational efficiency and for simple incorporation into the alternate tree structure. A node's *d-parent* ('dependency parent') is its parent in this tree. Nodes also depend on a nearest *head*, the most grammatically significant of the set of children of a given node in the parse tree. A node's *head* is then the closest such head on which it depends. The dependency tree can be defined in terms of the parse tree according to several simple rules which define the head and d-parent of a node.

Definition 7.5 (head, d-parent). The *head* and *d-parent* of a node are defined in terms of the parse tree as follows: (i) each non-leaf node has exactly one *head child*. If a node is a head child, it is said to be a head. If not, it is *non-head*. The root of the tree, representing the sentence, is always a head and is termed the *sentence head*; (ii) a non-head node's d-parent and head is that sibling (in the parse tree) which is a head; (iii) a head node's d-parent is its parent (in the parse tree). This may or may not be a head node itself. A head node's head is either its parent (where the parent is a head node) or its parent's head (where the parent is not a head node itself); (iv) a sentence head's d-parent is itself.

N is a *d-child* of N' if N' is a *d-parent* of N. A simple method is used to determine which child is a node's head child in the parse tree. A series of *headlists* $\{H_1, ..., H_n\}$ are defined. Each headlist H_i is associated with a class C_i, and a set of classes $H_{i,j}$. For each node N in the tree, the algorithm finds the first C_i such that N is of class C_i. The children of N are checked for membership of $H_{i,1}$, $H_{i,2}$, ... until some $H_{i,j}$ matches. The first child of class $H_{i,j}$ is then the head child. If no $H_{i,j}$ matches, the next headlist is checked. In the case where no headlist matches, a global headlist H_0 is used instead, which is assumed to match any class. Where no $H_{0,j}$ matches, the first child is chosen as the head child.

7.4.4 Relational structures in the SUSANNE corpus

A number of structures exist in the SUSANNE corpus. We illustrate the more important structures in the data with the aid of an example sentence from the corpus. The text of the sentence is as follows:

> The race problem has tended to obscure other less emotional issues which may fundamentally be even more divisive.

Figure 7.2 shows a detailed view of the relational structure of the sentence in the SUSANNE corpus. The structure comprises a *node* part and a *word* part. The node part consists of phrases (single-bordered boxes) and clauses (double-bordered boxes) arranged into a tree, in which child nodes are ordered. Each node is illustrated with its SUSANNE tag and the word, or range of words, to which it relates. Before the colon in the tag is the *form* or grammatical category and after the colon is the *function* or semantic role. Each node may have an (ordered) set of words associated with it, which appear with their SUSANNE part of speech tag. Words are linked to their

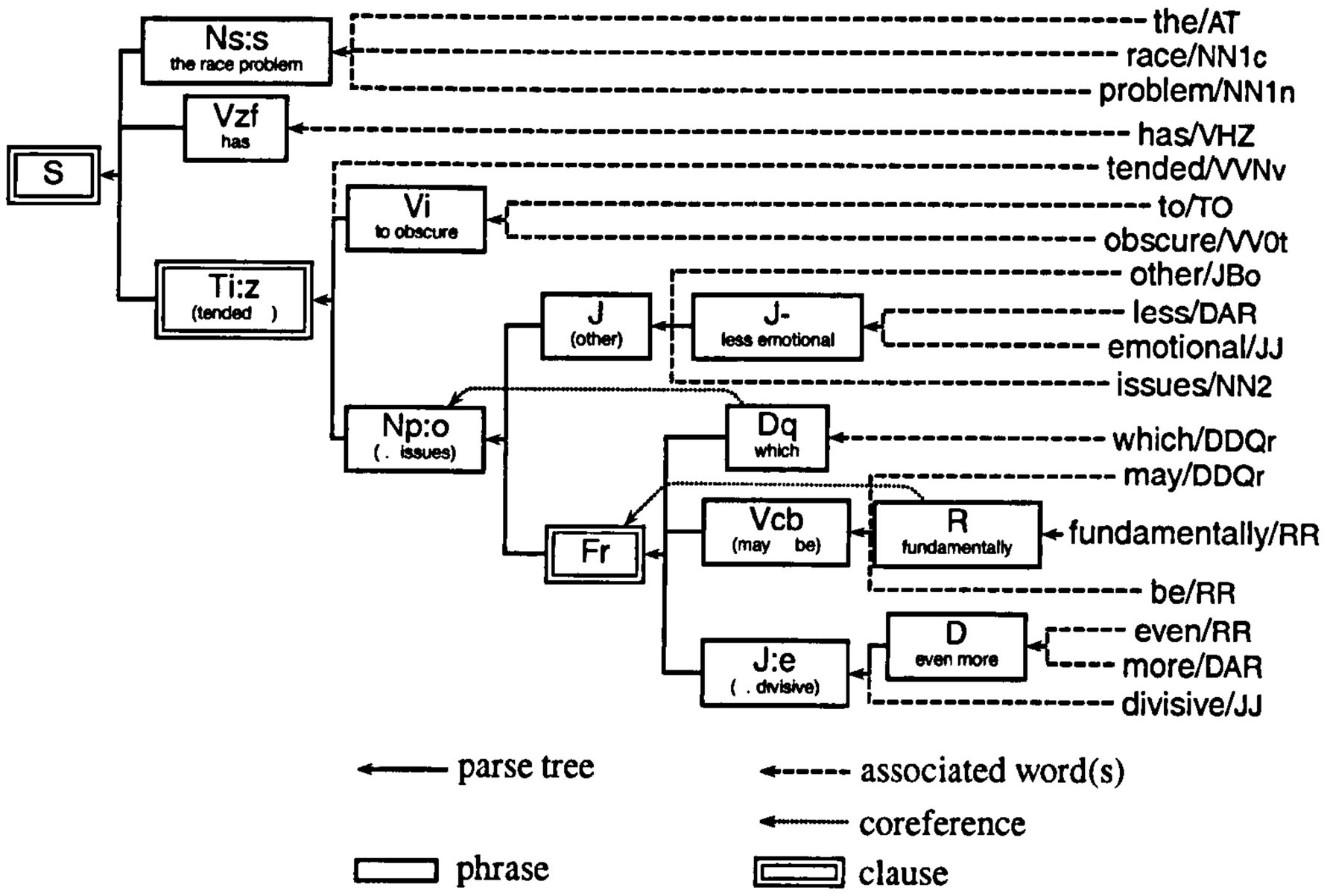
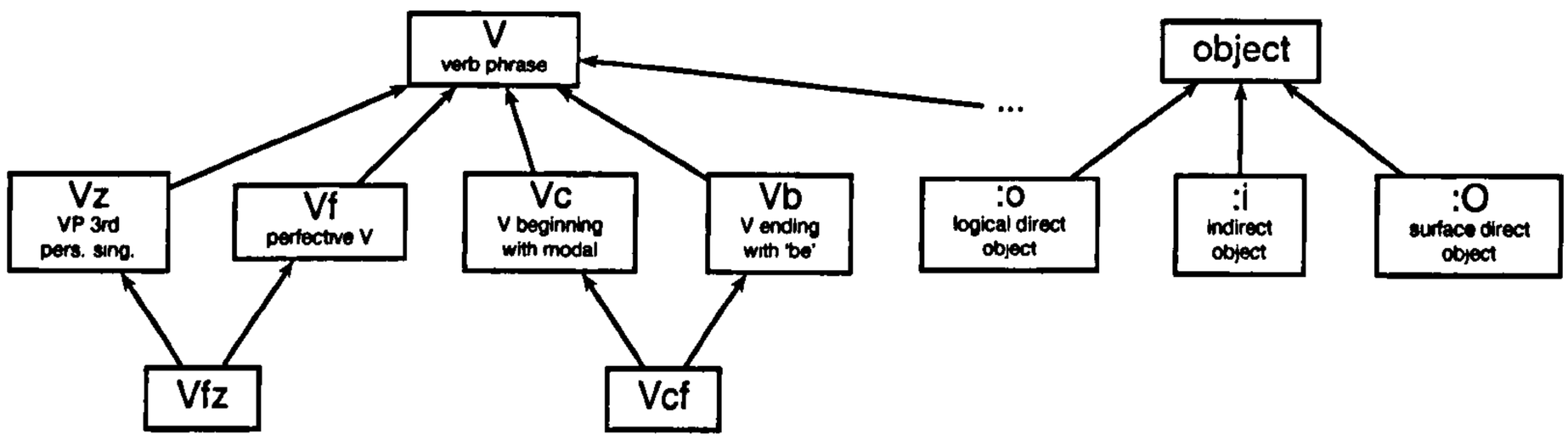


Figure 7.2: Compositional structure: A detailed view of the relational structure of a sentence in the SUSANNE corpus.



To the left, the inheritance lattice among form tags. To the right, an example of inheritance between function tags. The full inheritance structure contains many more classes representing each form and function tag.

Figure 7.3: Inheritance structure: A fragment of the inheritance hierarchy

preceding or following word. Coreference links appear in addition to the tree structure. To summarise, the data set may then be said to possess *four* main relational structures — the parse tree, the dependency tree, the list structure among words, and and the coreference links among clauses and phrases.

SUSANNE tags are translated by MAKETREE into database facts. Table 7.2 shows examples of the translation scheme on tags shown in figure 7.2. We consider the tags used in the examples here, describe their meaning, and provide the translation into Prolog and CORLOG. Grammatical categories appear as classes in the CORLOG representation and unary predicates in the Prolog representation. They may be specified by using subclasses, or by assigning properties to the classes.

Tag	*Meaning*	*Prolog translation for node id* o	
D	determiner phrase	`determinerphrase(o).`	o : determinerphrase.
Dq	wh-determiner phrase	`determiner_wh(o). wh(o,wh_).`	o : determiner_wh. o[wh → wh_].
Fr	relative clause	`relativeclause(o).`	o : relativeclause.
J	adjective phrase	`adjectivephrase(o).`	o : adjectivephrase.
J:e	adjective phrase as a subject complement	`adjectivephrase(o).` `subjectcomplement(o).`	o : adjectivephrase. o : subjectcomplement.
J-	adjective phrase as a non-conjunctive subordinate	`adjectivephrase(o).` `nonconjunctivesubordinate(o).`	o : adjectivephrase. o : nonconjunctivesubordinate.

Np:o	plural noun phrase as a logical direct object	`nounphrase(o).`
		`plurality(o,plural).`
		`logicaldirectobject(o).`
			o : nounphrase.
			o[plurality → plural].
			o : logicaldirectobject.

Ns:s	singular noun phrase as a logical subject	`nounphrase(o).` `logicalsubject(o).`	o : nounphrase. o : logicalsubject.
R	adverb phrase	`adverbphrase(o).`	o : adverbphrase.
S	main clause	`mainclause(o).`	o : mainclause.
Ti:z	infinitival clause as a catenative complement	`infinitivalclause(o).` `catenativecomplement(o).`	o : infinivalclause. o : cantenativecomplement.

Vcb	modal verbgroup (c), ending in infinitive 'be' (b)	`verbgroup_b_modal(o).`
		`beverbgroup(o). ending(o,be).`
		`itense(o,inf).`
			o : verbgroup_b_modal.
			o : beverbgroup.
			o[ending → be].
			o[itense → inf].

Vi	infinitival verb group	`infinitivalverbgroup(o).`	o : infinitivalverbgroup.
Vzf	third-person singular 'be' (z), perfective (f)	`verbgroup_b_3s(o).` `perfectiveverbgroup(o).`	o : verbgroup_b_3s. o : perfectiveverbgroup.

Combined form and function tags (*e.g.* Ns:s) are constituted from their form tag, before the colon, and function tag, after the colon. Tags may combine with each other. Translations are done at the most specific level possible. In the case of the tag Fr, were this tag F, the more general fact `f_clause(o).` would be translated, reflecting F's more general status compared to Fr.

Table 7.2: Examples of MAKETREE's translation

As well as this compositional structure, in which one node is associated with a set of constituent nodes, the tags form an inheritance, or subclassing, hierarchy. Figure 7.3 shows a fragment of a lattice representing this relationship. The full structure contains many classes representing each form and function tag. Two orthogonal structures result; the compositional structure of the parse tree, and the type lattice of the tag hierarchy.

We illustrate the use of the dependency hierarchy by introducing a typical set of headlists and presenting the resulting dependency hierarchy from the sentence shown in figure 7.2. Table 7.3 shows a simplified set of headlists, used for determining whether a node in the parse tree is the head child of its parent. The dependency structure is shown in figure 7.4. One child of each non-leaf node in the tree's compositional structure has been selected as its head child (shown shaded). Nodes are therefore head or non-head. An alternative structure results in the form of the dependency tree. Each node has one parent, and one dependant. Where the dependant differs from the parent, it is shown with a dashed line. Dependants occur as a result of nonmain/main groups, shown dotted in the tree.

Class	*Heads*
(global)	verb, noun, pronoun*, preposition, adverb, adjective
subordinate clause	verb, preposition
relative clause	verb, preposition
subordinate clause	coordinate*, verb, noun, pronoun, preposition
adjective phrase	adjective
adverb phrase	adverb
noun phrase	noun
verb phrase	verb

Each category in the head list is considered first as the class of a node and then as the (part-of-speech) class of a node's associated word unless marked with an asterisk, in which case the part-of-speech only is considered.

Table 7.3: A simplified set of headlists

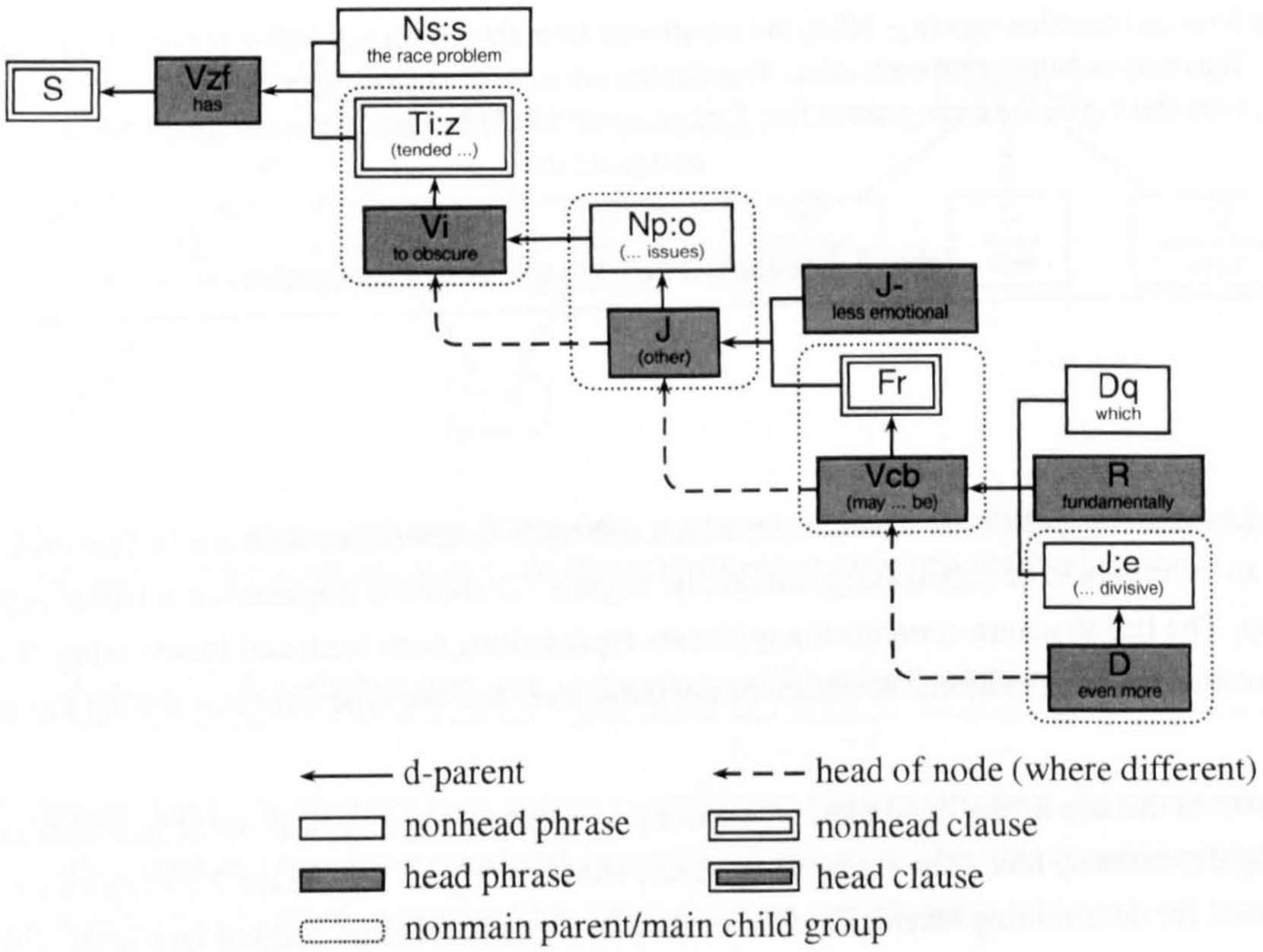


Figure 7.4: Dependency structure induced by application of the headlists.

Mapping the scheme into an object representation

With regard to the representation in the database, figure 7.2 can be considered as a set of objects, each of which contain a series of other objects. Child nodes of each parent node are ordered and are represented in the dataset by means of a relation linking them. Grammatical categories in the previous example are represented by classes in the CORLOG database. For example, the object *o* for the phrase THE RACE PROBLEM is associated with a formtag Ns and a function tag s. The form tag Ns corresponds to a noun phrase whose plurality is singular. The function tag s signifies that the phrase is a (logical) subject of its clause. The conversion script constructs the object *o* by asserting it as the member of nounphrase and logicalsubject. The phrase THE RACE PROBLEM is associated with *o*. An object-compositional structure is therefore derived from the parse tree and associated information in the corpus.

Inheritance relationships are defined by the coding scheme associated with the corpus [124]. The top part of the figure shows a fragment of the verb phrase inheritance hierarchy, in which arrows are drawn from a subclass to its superclass. In order to apply this to object-oriented data mining, we must consider how to map this into the object model. From this point of view, two features are of note. Firstly, the classes subclassing from the class associated with the formtag V may be interpreted either as classes in their own right, or as verb phrases associated with properties such as 'beginning with modal verb'. The choice of whether a formtag is refined in the hierarchy by either including a new subclass or by defining a property for the more general class is an important knowledge representation issue. Objects with formtag Vz, for example, could include appropriate values for a person and plurality property, constructing, for example, a fact o[plurarity → singular].

This would only be appropriate if such properties have a meaningful value for *all* verb phrases. In SUSANNE, this is not always known, and so here the structure implements them with classes. Secondly, formtags may combine to produce formtags such as Vfz, representing a perfective verb with a third person singular verb. Although this is meaningful and supported in the object model, such classes are not separately defined, but instead we allow nodes to belong to more than one class (provided disjointness conditions are respected). The lower part of the figure presents a similar inheritance hierarchy involving the function tags. Here we note that an object is a superclass of three function tags describing types of object. Although function tags are generally simpler than form tags, they may be abstracted into more general groups. Additionally, and not displayed, are function tags for other semantic roles, as well as adjunct tags denoting roles such as place, direction, time and manner.

7.4.5 The learning task

The learning task attempts to assign simple semantic roles to phrases and words in a clause. Many English sentences involve a head, or main, verb. Phrases and clauses which are dependent on this verb take semantic roles described by the function tag. Of particular interest is the relationship of a subject with its verb and words taking other grammatical categories. In principle, the other roles defined by the SUSANNE corpus could be tested, but for the purposes of these experiments we concentrate on the subject role only. Such a model has practical applications. The induced model provides a reliable means of labelling words in a sentence as possessing a given semantic role. This kind of analysis is of great importance in the study of machine translation. Since where we are able to automatically label nodes in a parse tree with a semantic tag in two languages, it far simplifies the task of correlated words in those languages. Furthermore, in areas such as document analysis, the search process can be aided by the labelling of semantic tags, enabling queries such as 'find all sentences

mentioning John Smith as the subject'. We aim to do this sort of analysis by ILP-induced classification rules, using the function tags in the data to classify nodes. The individual is then the node being tested. We refer to this node as the *chosen node*, aiming to induce classification rules which determine whether *any given node* in a sentence plays the part of a subject. We next describe how test data is generated. A sample size of 300 (150 positive and 150 negative examples) was used for the experiments, each example representing a distinct sentence from the corpus, by uniformly random selection among those sentences containing a subject node from the entire corpus. The dataset is thereby individual-centred, adopting the chosen node as the individual and ensuring individuals are not related. The positive examples are formed by uniformly random selection of 150 examples from this sample and labelling the node in the parse tree representing a subject nearest the root node as the chosen node. This node is then a positive individual. Among the remaining 150 examples, a node was unformly randomly selected from those not representing a subject as a negative example.

7.5 Experimental results

We have modelled the data from the SUSANNE corpus according to the object data model such that it can be reasoned about in the CORLOG logic and theories induced using the COSINUS system. We use this linguistic domain, and in particular the task of semantically labelling nodes in the parse tree as subject nodes, as a showcase for the object logic and induction process described in this thesis. The comparison follows three investigations of interest. Firstly we wish to understand how the learning parameters associated with COSINUS affect key aspects of the learner's performance. Primarily we are interested in the predictive performance of the learner, against its search complexity. We define this in terms of both the number of features searched as well as the running time. Additionally, we are interested in the degree to which clauses are searched by PROGOL which would be considered invalid under COSINUS. Secondly, we wish to understand how the presence or absence of object-oriented features in the learner affect these aspects. Finally, we use the datasets generated during the process of propositionalisation under these various settings in order to determine the result of applying REFER to these datasets.

PROGOL is adopted as a benchmark learner, and is included using comparable background knowledge under the mappings previously identified in this chapter. The version of PROGOL used is an adapted version of CProgol 4.4 which reports its search through the hypothesis space. Specifically, it reports each clause tested and whether the clause was accepted or rejected under the f_s criterion discussed earlier in section 7.1. Additionally, it records each refinement of a clause during search, and the clause it was refined from. In every other respect the learner is the same as CProgol 4.4. This additional output is parsed and summarised by a tracer program which provides statistics on the number of attempted and accepted refinements. COSINUS records corresponding information during its search which is compared with the PROGOL search statistics.

We apply the experimental method developed in section 7.3 to the computational linguistics task. Two levels of background knowledge are adopted for feature construction and learning, numbered B_1 and B_2. B_2 is used as the baseline. Table 7.4 summarises the predicates and method used in these settings.

In order to understand how settings and variants affect performance, we adopt a base learner, deviating from it in selected parameter settings. Table 7.5 summarises the settings adopted for these experiments, each characterised by a short name, for both the COSINUS and PROGOL learner. COSINUS parameters investigated consider the maximum number of applications of the refinement operator during rule search, the maximum number of methods in the relational part of a CORLOG feature; the size of a conjunctive class; and the number

Setting	*Predicate or method*	*Description*
C_1	`containscp`(+`node`, −`cp`) node[containscp ⇒ cp]	Succeeds if a node representing a clause or phrase is beneath the given node in the parse tree.
	`tnext`(+`node`, −`node`) node[tnext ⇒ node]	Succeeds if the output node is treewise next to the input node.
	`tprev`(+`node`, −`node`) node[tprev ⇒ node]	Succeeds if the output node is treewise previous to the input node.
	`withinscope`(+`node`, −`node`) node[withinscope ⇒ node]	Succeeds if the output node is the parent or sibling of the input node.
	grammatical categories	Unary predicates and class declarations representing various grammatical categories and the hierarchical relationships between them.
C_2	`dependent`(+`node`, −`node`) node[dependant ⇒ node]	Succeeds if the input node is *directly* dependent on the output node.
	`dchild`(+`node`, −`node`) node[dchild ⇒ node]	Succeeds if the output node is a d-child of the input node.
	`containsword`(+`node`, −`word`) node[containsword ⇒ word]	Succeeds if the output word appears at or under the input node in the parse tree.
	part-of-speech categories	Unary predicates and class declarations representing parts of speech for words and their higher-level categories.

Background knowledge is cumulative: $B_1 = C_1$, $B_2 = B_1 \cup C_2$.

Table 7.4: Summary of background knowledge categories used in the experimental evaluation.

Experiment	*System*	*Background knowledge*	*Max. refinements*	*Max. literals*	*Coverage bound*	*Max. class conj. length*	*Object identity*	*Multiplicity*	*Abstract/disjoint classes*	*Merging in MGCS*	*Background knowledge*	*Max. literals*	*Mapping adopted*
base	CoSinus	B_2	4	2	3	1	on	on	on	on	—	—	—
bg1	CoSinus	$\mathbf{B_1}$	4	2	3	1	on	on	on	on	—	—	—
2l/5a	CoSinus	B_2	**5**	2	3	1	on	on	on	on	—	—	—
3l/5a	CoSinus	B_2	**5**	**3**	3	1	on	on	on	on	—	—	—
cov=1	CoSinus	B_2	4	2	**1**	1	on	on	on	on	—	—	—
cov=0	CoSinus	B_2	4	2	**0**	1	on	on	on	on	—	—	—
nooid	CoSinus	B_2	4	2	3	1	**off**	on	on	on	—	—	—
nomgcs	CoSinus	B_2	4	2	3	1	on	on	on	**off**	—	—	—
nomult	CoSinus	B_2	4	2	3	1	on	**off**	on	on	—	—	—
cc2	CoSinus	B_2	4	2	3	**2**	on	on	on	on	—	—	—
cc2-nd	CoSinus	B_2	4	2	3	**2**	on	on	**off**	on	—	—	—
cc2-nm	CoSinus	B_2	4	2	3	**2**	on	on	on	**off**	—	—	—
e1 base	Progol	—	—	—	—	—	—	—	—	—	B_2	4	E_1
e2 base	Progol	—	—	—	—	—	—	—	—	—	B_2	4	E_2
e1 bg1	Progol	—	—	—	—	—	—	—	—	—	$\mathbf{B_1}$	4	E_1
e2 bg1	Progol	—	—	—	—	—	—	—	—	—	$\mathbf{B_1}$	4	E_2
e1 3l	Progol	—	—	—	—	—	—	—	—	—	B_2	**3**	E_1
e2 3l	Progol	—	—	—	—	—	—	—	—	—	B_2	**3**	E_2

Changes to parameters from base settings are show in bold.

Table 7.5: Summary of parameter settings for each experiment.

of examples which must be covered by a feature in order for it to remain in the search. With respect to the last point, a set of ten seed examples is chosen over which to estimate this coverage. Variants of the learner are obtained by enabling and disabling object identity, multiplicity, the process of type specialisation during refinement, the presence of abstract and disjoint classes, and the handling of unification of combination classes (whether merging is permitted in determining a unifier). The concept of disjoint classes is only useful where more than one class can be assigned to the same term. Accordingly, we consider its removal only by comparison with combination classes. PROGOL is adopted as a baseline of traditional ILP, being a well-established and proven system. After propositionalisation, REFER is applied for each experiment on the transformed data, using the feature ranking variant. We discuss the effects of this later in section 7.5.3.

In general, the data points in the graphs summarise values gained over a ten-fold cross validation. We then calculate the mean and standard error of the measurements resulting from each fold. Data points in each graph shown show the mean over these ten folds, and the bars the standard error. Results are reported numerically in tables 7.6, 7.7 and 7.8. Firstly, we consider the feature search, characterising it in three ways; the total number of features searched, the number supported — those that are found to be of sufficient quality by the refinement operator — and, in the case of COSINUS, the number of features selected for propositionalisation. PROGOL takes an iterative approach, and so mean sizes per iteration are considered, as well as the total number of iterations taken. Features supported and selected are also reported as a percentage of the total size of the feature search. This is in spite of the fact that the total number of features searched (and checked against the examples according to the heuristic) is many times greater. What we intend to do by considering the iteration-level statistics is to suggest the general size of the search space explored by PROGOL. The number of duplicate and metaknowledge-invalid features in the PROGOL search are also considered, both over the whole search and the average within each iteration. These are given as a percentage of the total features search, either overall or per iteration. The area under the ROC curve (AUC) over each fold is reported. Finally, the running time of the algorithm in seconds is given. In all experiments, REFER was applied to the propositionalised data before it was supplied to the CN2 algorithm, although later in section 7.5.3 we consider the effect of doing this in isolation.

7.5.1 Comparing parameter settings for COSINUS

We first turn our attention to the numerical data in table 7.6 and identify the key results they demonstrate. Later we consider the variant forms of COSINUS. In this comparison we also present results for six main variants of the PROGOL learner, described by the form of domain definition which they use, the set of background knowledge used, and the number of atoms permitted in its clause body.

We discuss general patterns in the data first before moving onto the effect of applying particular parameters on the results reported. Firstly, perhaps the most striking result is that owing to the number of iterations, the total number of features searched by PROGOL is far greater than those of COSINUS. This is partly due to the iterative approach taken by PROGOL. The resulting search is computationally more complex, since PROGOL uses a heuristic which considers the coverage of each feature over positive *and* negative examples. COSINUS instead is required to construct one set of features for the whole example set in its single-iteration mode, but in doing so tests each feature against each example in its seed set only once, leading to preferable complexity. The high number of iterations taken by PROGOL suggest that each rule found on average covered few examples. in practice, PROGOL often could not generalise an example sufficiently to form a rule. We see this effect later in the combined ROC curve for each fold. In terms of accuracy, the base learner achieves an AUC of slightly under

System	Experiment	Features searched	Features supported	Features selected	Test AUC	CPU time used (s)
CoSinus	*base*	329.9 ± 17.7	110.8 ± 5.4 (33.6%)	72.6 ± 3.9 (22.0%)	0.898 ± 0.016	91.0 ± 3.6
	bg1	234.3 ± 7.1	75.4 ± 2.4 (32.2%)	49.2 ± 2.1 (21.0%)	0.799 ± 0.019	51.0 ± 2.0
	2l/5a	432.3 ± 22.2	125.2 ± 6.5 (29.0%)	81.0 ± 4.8 (18.7%)	0.897 ± 0.017	108.1 ± 5.1
	3l/5a	1165.2 ± 55.5	393.7 ± 16.6 (33.8%)	252.4 ± 12.8 (21.7%)	0.879 ± 0.018	284.9 ± 13.0
	cov=1	484.4 ± 12.8	253.3 ± 9.2 (52.3%)	201.0 ± 8.1 (41.5%)	0.896 ± 0.011	170.2 ± 5.3
	cov=0	605.0 ± 0.0	605.0 ± 0.0 (100.0%)	549.0 ± 0.0 (90.7%)	0.880 ± 0.018	318.0 ± 1.4
Progol	*e1 base*	14186.6 ± 479.0	8727.8 ± 246.2 (61.5%)	—	0.625 ± 0.014	140.6 ± 5.4
	e2 base	11068.7 ± 322.4	6708.2 ± 179.3 (60.6%)	—	0.850 ± 0.029	90.6 ± 1.8
	e1 bg1	19952.9 ± 491.5	11857.3 ± 232.5 (59.4%)	—	0.643 ± 0.017	114.3 ± 3.9
	e2 bg1	18364.4 ± 341.6	11367.1 ± 208.4 (61.9%)	—	0.675 ± 0.020	78.3 ± 1.8
	e1 3l	14141.1 ± 390.1	9036.2 ± 189.3 (63.9%)	—	0.644 ± 0.017	130.7 ± 3.5
	e2 3l	12926.2 ± 307.1	7895.3 ± 183.4 (61.1%)	—	0.843 ± 0.028	103.3 ± 2.4

System	Experiment	Iterations	Features searched per iteration	Features supported per iteration	Dup. feats. inside each it.	Duplicate features	Invalid features
Progol	*e1 base*	43.6 ± 1.2	325.4 ± 5.9	200.2 ± 0.0 (61.5%)	108.4 ± 0.5 (33.3%)	7501.2 ± 225.5 (85.9%)	748.4 ± 28.4 (8.6%)
	e2 base	33.5 ± 0.9	330.3 ± 2.0	200.2 ± 0.0 (60.6%)	102.4 ± 0.7 (31.0%)	5480.1 ± 185.6 (81.7%)	56.4 ± 5.5 (0.8%)
	e1 bg1	59.2 ± 1.2	336.9 ± 3.8	200.3 ± 0.0 (59.5%)	104.5 ± 0.4 (31.0%)	10743.7 ± 228.7 (90.6%)	1712.1 ± 46.0 (14.4%)
	e2 bg1	73.1 ± 1.1	251.2 ± 1.7	155.5 ± 0.8 (61.9%)	69.0 ± 0.6 (27.5%)	10449.0 ± 190.9 (91.9%)	47.6 ± 5.4 (0.4%)
	e1 3l	45.1 ± 0.9	313.3 ± 3.5	200.4 ± 0.0 (64.0%)	103.0 ± 0.6 (32.9%)	8367.3 ± 195.8 (92.6%)	653.6 ± 18.8 (7.2%)
	e2 3l	39.5 ± 0.9	327.3 ± 2.3	199.9 ± 0.0 (61.1%)	95.9 ± 0.5 (29.3%)	7269.5 ± 180.5 (92.1%)	57.8 ± 3.3 (0.7%)

± denotes standard error over ten folds. Features supported and selected are also shown as a percentage of the total number of features searched. Percentages for duplicate and invalid features are in terms of the number of features searched.

Table 7.6: Summary of results for varying parameter settings

0.9, compared to an AUC of 0.85 for the best PROGOL learner. We shall see later that this AUC is improved upon further by the variant forms of CoSINUS.

Observations about the search may also be made. Excepting the special case of *cov=0*, CoSINUS's proportion of features supported to features searched is much lower than for PROGOL. This suggests that in general CoSINUS is more selective in its refinement than PROGOL. Following this, given the predictive power of the theory, the number of features selected for propositionalisation is very small; for the base learner only 72 features were sufficient for CN2 to construct a well-performing rule. Related to the selectivity of the refinement operator, note that PROGOL is bounded above by approximately 200 in its number of features supported per iteration. This is a direct consequence of the default nodes setting of 200. PROGOL's refinement operator is more selective, and therefore places a resulting bound on the number of nodes which it searches in total. Despite this, the number of nodes searched *per iteration* is comparable to CoSINUS's base setting.

In general, the running time of CoSINUS's base learner, 91s, is favourable compared to the base learners

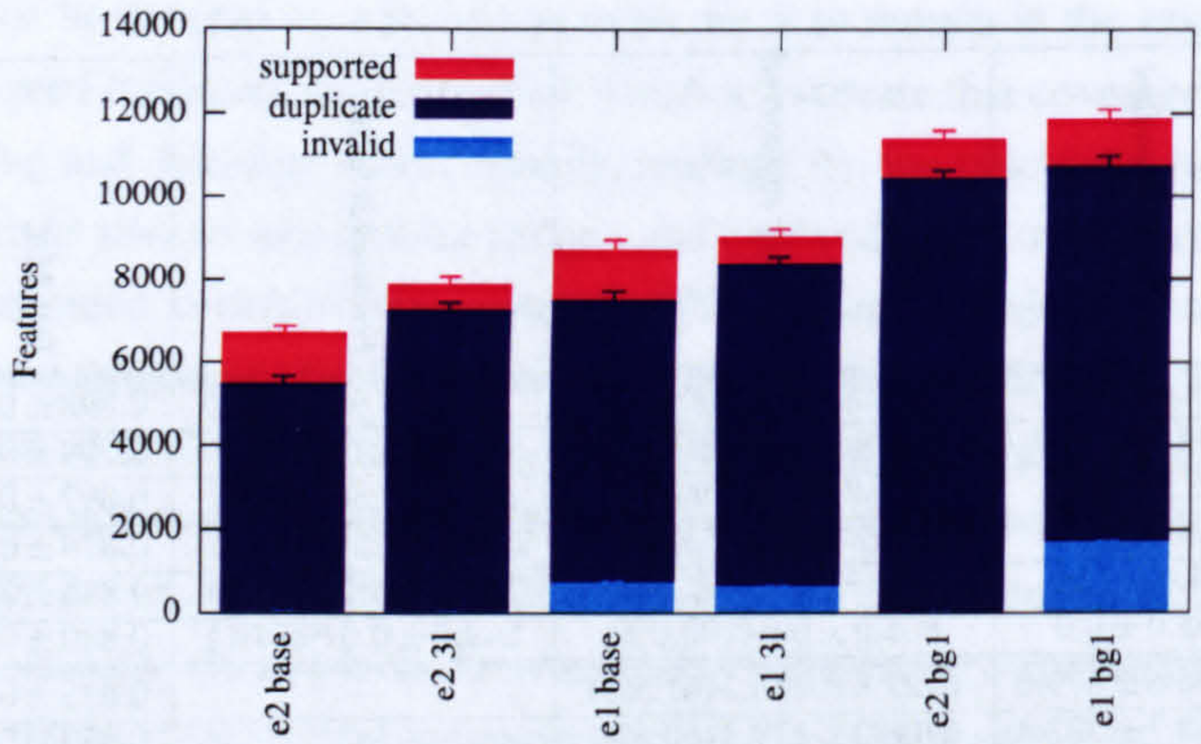


Figure 7.5: Duplicate and invalid features over all iterations

of PROGOL, which took approximately the same time (under background knowledge mapping E_2) or much longer (under E_1), and COSINUS achieved better predictive performance. Considering the duplication results, PROGOL appears to revisit the same clause repeatedly on successive iterations. Considering a single seed example, as PROGOL does, might lead it to yield a widely differing search space on successive iterations, but a high proportion of features are repeated per over each iteration, meaning only a small fraction of features explored were unique to their iteration. *Within iterations*, PROGOL's refinement operator searches a considerable proportion of duplicate examples. It therefore exhibits undesirable redundancy. Furthermore, the proportion of metaknowledge-invalid features is relatively high over the full feature search in the case of the E_1 mapping, which shows that unless a specific hypothesis space reduction measure such as E_2 is adopted in the background knowledge, simplifying class hierarchy drastically by resituating them as properties of a object, metaknowledge-invalid features will result. Indeed, it should be noted that the presence of metaknowledge-invalid features is highly domain-dependent. This comparative analysis necessarily used a simple class model. Under a more complex model it would be likely that many more metaknowledge-invalid features would result.

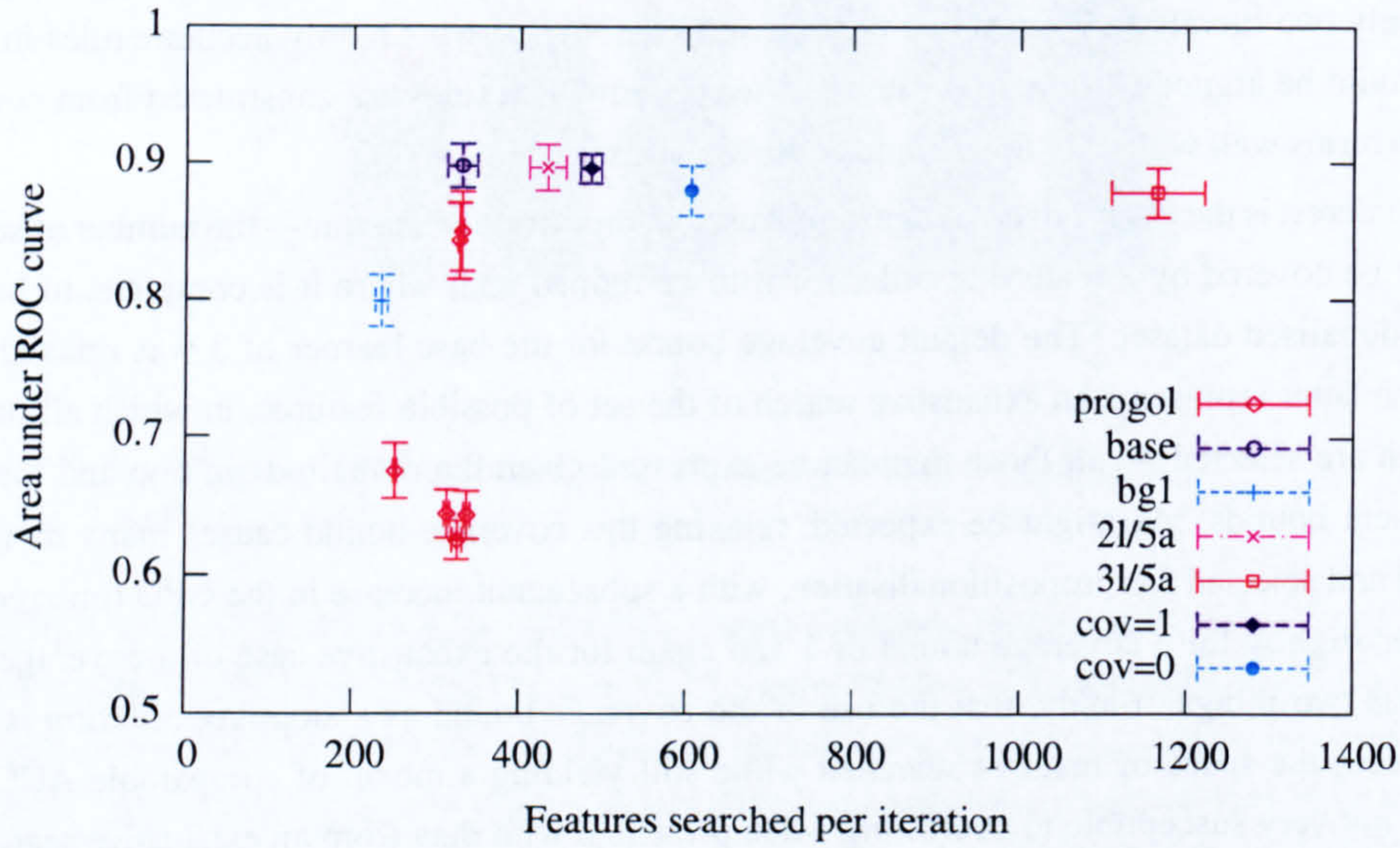
This metaknowledge-invalidity and duplication of features among iterations is illustrated graphically in figure 7.5. It supports our claim that PROGOL searches a considerable proportion of metaknowledge-invalid features, which under COSINUS would not be considered. It suggests that COSINUS's use of metaknowledge to constrain the search space is an effective means of bounding feature generation. Returning to the point regarding the nature of the mapping, note that the mapping E_2, in which class membership is modelled as a property of an object with an integer symbol, shows a lower degree of invalidity compared to E_1, in which each subclass is represented with a unique, typed, predicate. Since each subclass membership predicate under E_2 takes a recall number of 1, only one subclass per object is introduced, and invalidity introduced by subclassing and disjoint classes is necessarily limited. Furthermore, metaknowledge-invalidity is expected to rise considerably for biases which permit more atoms in the body of the rule, since these atoms introduce constraints on terms, among which much of the metaknowledge is defined.

We now consider the effect of parameter settings on the performance of COSINUS, and, to a lesser extent, PROGOL. Firstly, we examine how the learner performs under simpler background knowledge. When B_2 is replaced with the background knowledge set B_1, there is a noticeable reduction in the AUC of approximately 0.2. Under COSINUS, this drop is less severe, at approximately 0.1. COSINUS's running time and search

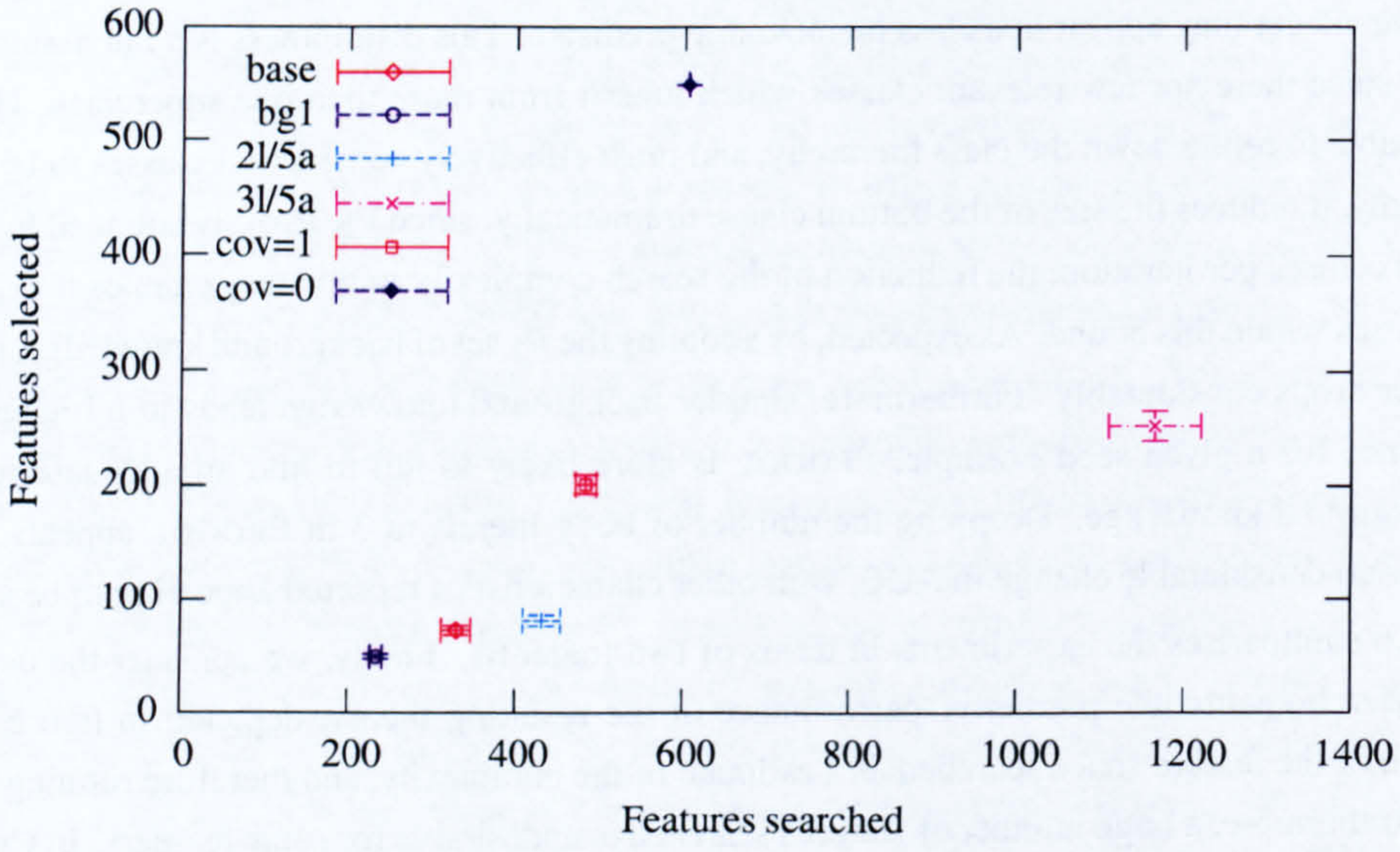
complexity drop considerably as a result, and only 49.2 features are selected on average for a theory which still possesses a high level of predictive power. Interestingly, increasing the number of refinements per feature (experiment *2l/5a*) does not seem to increase the size of the feature search, and yields no greater predictive power. A similar result follows for three literals for feature and five refinements (*3l/5a*), indicating that features involving only two literals and four refinements is sufficient to construct highly-accurate rules in this domain. Indeed, it could be argued that the multi-levelled search, in which rules are constructed from conjunctions of features, performs well with only few simple features in this domain.

Also of interest is the choice of coverage bound used in propositionalisation — the number of seed examples which must be covered by a feature in order for it to be refined and, where it is complete, to be included in the propositionalised dataset. The default coverage bound for the base learner of 3 was relaxed to values of 1 and 0. The latter represents an exhaustive search of the set of possible features, in which all of the features in the search are selected — all those that can be expressed given the domain definition and the feature size and refinement bounds. As might be expected, relaxing this coverage bound causes many more features to be searched and selected for propositionalisation, with a subsequent increase in the CPU time required. AUC results lower slightly for a coverage bound of 1 and again for the exhaustive case of a coverage bound of 0. This suggests two things. Firstly, that the use of the coverage bound as a stopping criterion is appropriate, since it reduces the space of features searched while still yielding a model of comparable AUC. Secondly, that CN2 is not very susceptible to overfitting when presented with data from an exhaustive search. We now briefly consider some effects in the PROGOL experiments. As mentioned before, the E_1 form of background knowledge produces much less successful theories than the E_2 form. One possible reason is that under the E_2 mapping, subclasses are intrinsically encoded as being disjoint as a result of the fact that with a recall number of 1, only one integer may appear in a class membership predicate. This disjointness is a fair assumption under this domain since there are few relevant classes which inherit from more than one superclass. However, the learner is unable to refine down the class hierarchy, and must effectively consider all classes to be at the same level. Secondly, it reduces the size of the bottom clause dramatically. since PROGOL is bounded by the number of supported clauses per iteration, the reduction of the search complexity in this way enables it to search more effective clauses within this bound. As expected, by adopting the B_1 set of background knowledge, the accuracy of the learner drops considerably. Furthermore, simpler background knowledge leads to a higher number of iterations, since for a given seed example, PROGOL is more likely to fail to find an adequate rule with the simpler background knowledge. Dropping the number of body literals to 3 in PROGOL appears to lead to a small drop or no considerable change in AUC, with other characteristics reported appearing to be similar also.

Figure 7.6 summarises the experiments in terms of two tradeoffs. Firstly, we consider the complexity of the feature search against the predictive performance of the resulting theory, depicted in figure 7.6(a). We adopt the size of the feature space searched as a estimate of the complexity, and therefore running time, of the algorithm. Furthermore, a large number of features searched is undesirable for some learners. In COSINUS, for example, a large number of features may introduce overfitting issues into the learner — the well-known curse of dimensionality. We therefore consider feature space searched as form of cost, relating it to the test of coverage of an example by a feature. Again, we consider PROGOL's feature space per iteration. Although PROGOL necessarily searches many more features over many iterations, it could feasibly cache the results of the coverage tests on the examples. With respect to the number of times a clause needs to be resolved against an example, in the *best* case, the same clauses appear in each iteration, and only the average number of clauses searched per iteration need to be tested against each example for coverage. The number thus introduces a lower bound (since



(a) AUC for number of features supported for varying parameters



(b) Number of features selected for number of features searched for varying parameters

Figure 7.6: Efficiency and complexity of search for varying parameters.

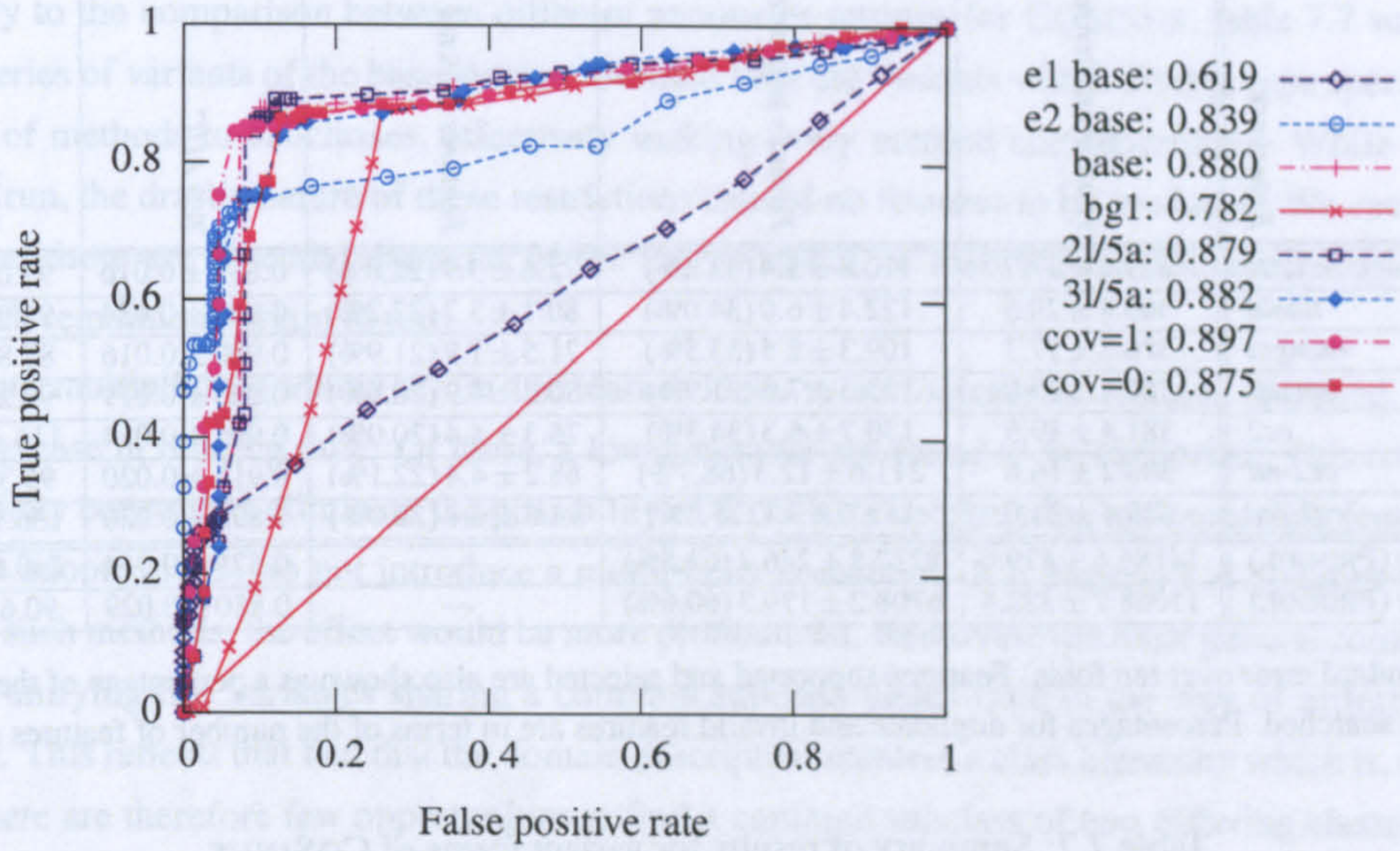


Figure 7.7: Combined ROC curve for varying parameters

clauses searched can differ over iterations) on its search complexity. Necessarily, CoSinus compares every example against every feature once in the propositionalisation (for the number of features selected) and each searched feature against the number of seed examples during search. Since this is not all possible features, the adoption of the number of features search provides a convenient upper bound on comparable search complexity.

Of immediate interest is the property that the points corresponding to the CoSinus learner all lie on the convex hull of the graph. Informally, this suggests that for any tradeoff between predictive performance and feature search, CoSinus represents the best learner, based on the results obtained. More formally, for a linear map $S(x,y) = -ax + by$ representing the desirability (defined by positive constants a and b) of a solution in x (the number of features searched) and y (the accuracy), the values of S for a given (x,y) representing CoSinus will be greater than one representing Progol. This verifies our first claim, that CoSinus searches fewer features for comparable or better predictive performance of the induced classifier.

The second graph in figure 7.6(b) depicts the number of features selected per number of features searched, and represents to what extent suitable features are found from searched of varying complexity. Except for the points representing various coverage settings, the points seem to exhibit an approximately constant rate of feature selection (the gradient of the best fit line through these points), with progressively more complex feature spaces (*2l/5a* and *3l/5a*) requiring larger searches.

Figure 7.7 continues the analysis of the predictive accuracy of the models obtained under various parameter settings by considering their ROC curves. The curve shown is a combination of the ROC curves obtained over each of the ten folds. Combination ROC curves are a suitable way of visualising the performance of a classifier across many folds. In order to plot a combination ROC curve, we consider each of the rules in all ten experimental folds, and for each of the rules, we determine its coverage on the positive and negative test and training data. We assign each rule a Laplace-corrected value $\frac{p+1}{n+1}$ if it covers p positive training examples and n negative training examples from the training data for its fold. Segments corresponding to each rule are then plotted decreasing order of this value, according to the test set coverage of the rule. *i.e.*, if the rule covered p' positive test examples and n' negative test examples, a straight-line segment covering p' units on the y-axis and

Experiment	*Features searched*	*Features supported*	*Features selected*	*Test AUC*	*CPU time used (s)*
base	329.9 ± 17.7	110.8 ± 5.4 (33.6%)	72.6 ± 3.9 (22.0%)	0.898 ± 0.016	91.0 ± 3.6
nooid	360.4 ± 20.5	122.4 ± 6.0 (34.0%)	80.1 ± 3.7 (22.2%)	0.840 ± 0.034	96.9 ± 4.7
nomgcs	326.2 ± 17.1	109.3 ± 2.5 (33.5%)	71.5 ± 1.9 (21.9%)	0.900 ± 0.016	87.8 ± 2.2
nomult	386.4 ± 25.0	123.3 ± 7.0 (31.9%)	80.5 ± 4.9 (20.8%)	0.893 ± 0.019	99.2 ± 4.3
cc2	381.4 ± 19.5	130.7 ± 6.3 (34.3%)	76.3 ± 4.4 (20.0%)	0.905 ± 0.014	116.4 ± 9.4
cc2-nd	308.2 ± 16.6	211.6 ± 12.3 (68.7%)	68.2 ± 4.4 (22.1%)	0.915 ± 0.020	95.3 ± 8.5
cc2-nm	429.4 ± 24.0	153.6 ± 9.1 (35.8%)	94.6 ± 6.1 (22.0%)	0.895 ± 0.020	136.1 ± 9.6
e1 base (PROGOL)	14186.6 ± 479.0	8727.8 ± 246.2 (61.5%)	—	0.625 ± 0.014	140.6 ± 5.4
e2 base (PROGOL)	11068.7 ± 322.4	6708.2 ± 179.3 (60.6%)	—	0.850 ± 0.029	90.6 ± 1.8

± denotes standard error over ten folds. Features supported and selected are also shown as a percentage of the total number of features searched. Percentages for duplicate and invalid features are in terms of the number of features supported.

Table 7.7: Summary of results for variant forms of CoSINUS

n' units on the x-axis would be plotted. Finally, the curve is scaled back to form a ROC curve. The area under each curve for the test data is shown in the key directly after the name of the experimental setting.

We consider all PROGOL experiments and all CoSINUS parameter settings explored in these experiments on the graph as separate plots. The curves for many of the CoSINUS parameters are very similar, represented by the cluster of lines which rise steadily, round off near $(0,1)$, and form an approximately co-linear sequence of segments toward $(1,1)$. The curve for *bg1* occupies a lower trajectory, but rejoins the other CoSINUS plots later. The initial segments represent those rules which have class distribution covering more positive examples, and the later segments covering more negative examples, the gradient of each segment being equal to the ratio of positive to negative test examples covered by the rule corresponding to the segment. The pointed nature of the curve suggests that CoSINUS induces many rules which cover a large number of one class and few of the other — there are few rules with an approximately equal number of examples covered. ROC curves for PROGOL, on the other hand, tend to rise rapidly in a sequence of small segments, keeping close to the y-axis, and then proceed, again in an approximately colinear way, to $(1,1)$. The reason for this general shape is the high number of specific positive rules which a typical PROGOL theory in this domain involves. The short initial segments represent small numbers of positive examples covered by a large number of rules, while the longer later segments represent the failure of any rule in the PROGOL theory to fire, classifying it as negative. Two trends exist for the PROGOL experiments, one taking the shape described, below the CoSINUS experiments, and corresponding to the E_2 mapping. Both the 3- and 4-literal settings follow the same shape, with the 3-literal setting appearing slightly beneath the 4-literal setting. The remaining PROGOL runs, which scored a low AUC, again assume the same shape, and appear closer to the random classifier line.

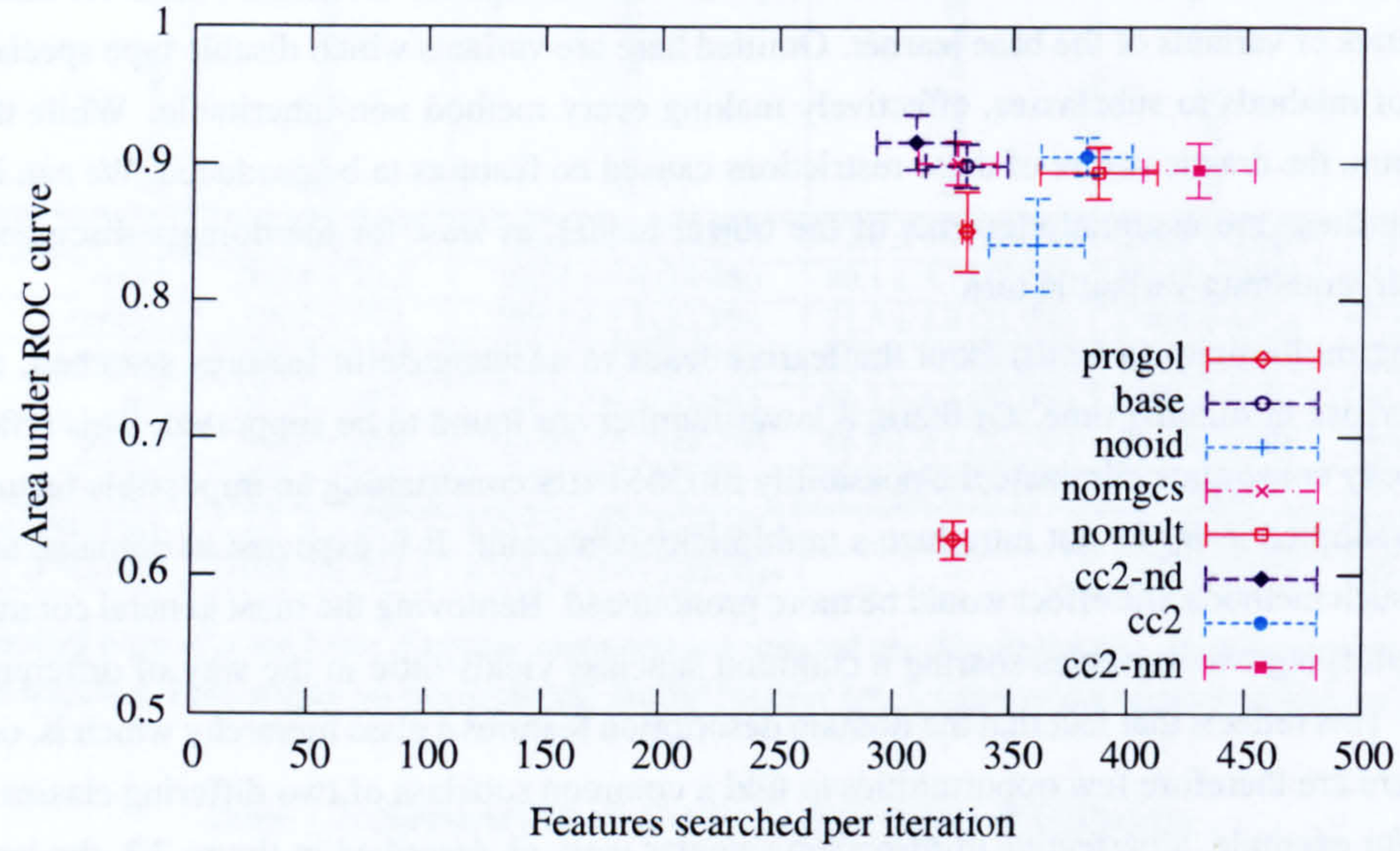
7.5.2 Comparing variant forms of CoSINUS

In order to understand the utility of the metaknowledge declarations in CoSINUS, we compare a series of variants introduced earlier in this chapter. For clarity, we retain the base learner E_2 of PROGOL only, having compared PROGOL results in the previous section.

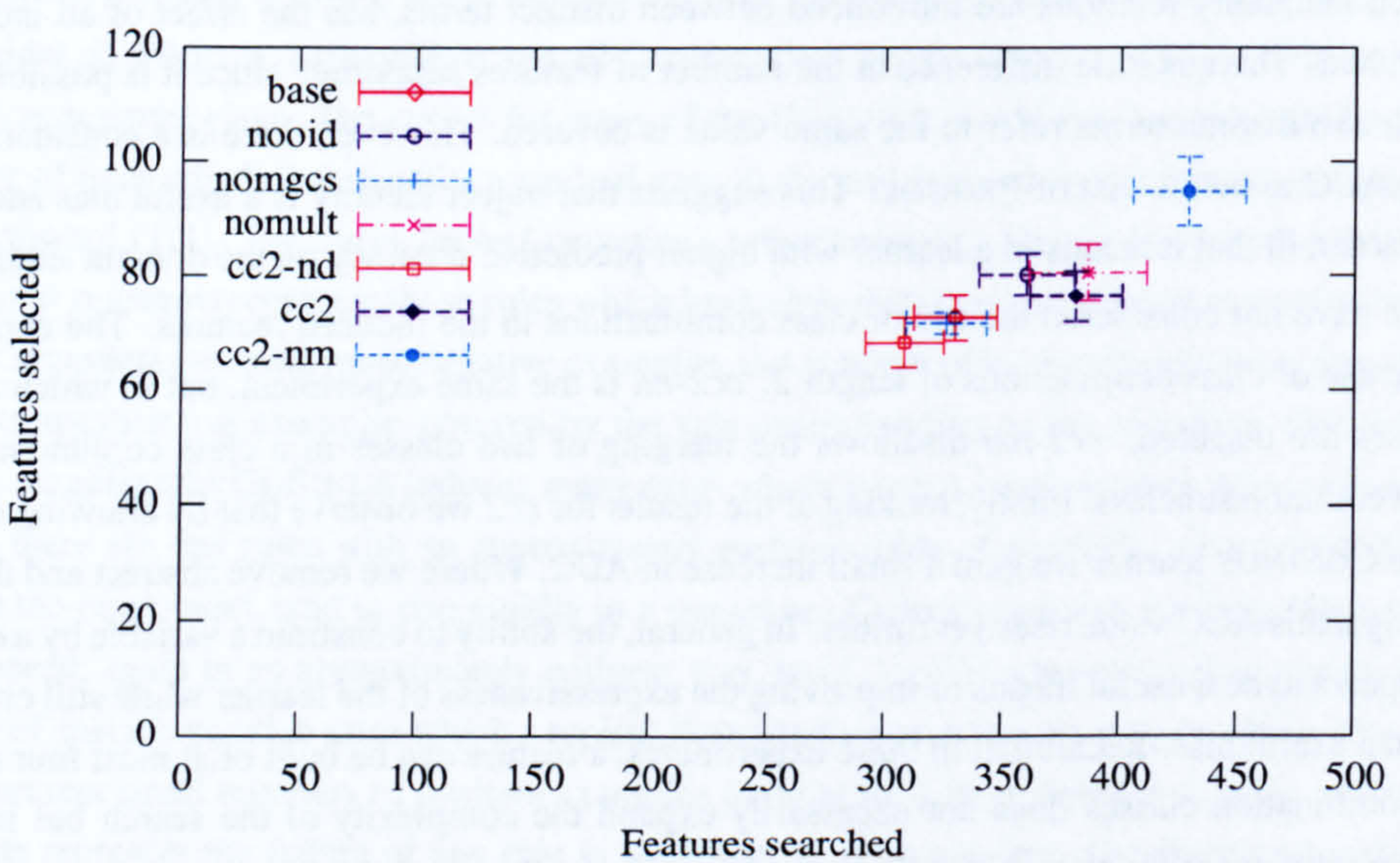
Similarly to the comparison between different parameter settings for CoSinus, table 7.7 summarises results for a series of variants of the base learner. Omitted here are variants which disable type specialisation and inheritance of methods to subclasses, effectively making every method non-inheritable. While these experiments were run, the drastic nature of these restrictions caused no features to be produced. We can immediately conclude that these are essential elements of the object model, at least for the domain discussed here. We compare each remaining variant in turn.

Removing multiplicity (*nomult*) from the learner leads to an increase in features searched, and a corresponding increase in running time. Of these, a lower number are found to be supported. This reflects the fact that multiplicity constraints eliminate the possibility of CoSinus constructing an impossible feature. Many of the methods adopted in B_2 do not introduce a multiplicity constraint. It is expected in domains that naturally incorporate such methods, the effect would be more pronounced. Removing the most general common subclass method for unifying two variables sharing a common subclass yields little in the way of difference from the base learner. This reflects that fact that the domain description features a class hierarchy which is, on the whole, treelike. There are therefore few opportunities to find a common subclass of two differing classes. Instead, in the cast of, for example, a perfective third-person singular verb, as described in figure 7.3, the learner simply constructs this as a conjunction of two classes, one for the perfective class and one for the third-person singular verb. This combination is naturally part of the setting for class combinations. Removing the object identity bias, in which inequality relations are introduced between distinct terms, has the effect of an increase in the features searched. There is little difference in the number of features supported, since it is possible that a feature in which two distinct terms refer to the same value is covered. However, there is a considerable drop in the resulting AUC to below that of Progol. This suggests that object identity is a useful bias adopted by the CoSinus learner, in that it results in a learner with higher predictive accuracy on the domain studied.

So far we have not considered the use of class combinations in the induced features. The experiment *cc2* considers the use of class conjunctions of length 2. *cc2-nd* is the same experiment, but in which abstract and disjoint classes are disabled. *cc2-nm* disallows the merging of two classes in a class conjunction into their most general common subclass. Firstly, looking at the results for *cc2* we observe that by allowing combination classes in the CoSinus learner we gain a small increase in AUC. Where we remove abstract and disjoint class metaknowledge, this AUC value rises yet further. In general, the ability to constrain a variable by a combination of classes appears to be a useful means of improving the expressiveness of the learner while still containing the search within a useful bias. Recall that in these experiments, a feature can be built of at most four refinements. Permitting combination classes does not necessarily expand the complexity of the search but instead tends to concentrate refinement on class specialisation. Accordingly, the number of features searched for *cc2* and variants is not considerably larger than that of the base learner, except in the case where merging is disallowed, since the learner is more likely to specialise class further, rather than substituting, thereby exploring deeper into the class tree. An interesting result is that for when abstract and disjoint classes are disabled; the number of features supported jumps to 68.7%. This suggests that class combinations useful for constructing features with high predictive power were found with fewer features searched. A possible explanation is that removing abstract classes and disjoint classes encourages the learner to specialise along the class hierarchy more, which may have a lesser branching factor that with other refinement operators, and it is the class membership constraints which prove most useful in learning. Another explanation is that the domain knowledge supplied to the learner, while correct with respect to commonly-accepted linguistic categories, did not reflect the structure of the data, due to corpus input errors, linguistic exceptions, *etc.*



(a) Predictive accuracy for number of features selected for CoSinus variants



(b) Number of features selected for number of features searched for CoSinus variants

Figure 7.8: Efficiency and complexity of search for CoSinus variants

In general, however, it can be seen that the omission of elements of the domain model lead to increases in the size of the feature set searched or reductions in the predictive power measured by the AUC, or both. On the other hand, the use of the combination class mechanism seems, on the whole, to lead to a increase in the predictive power of the model with no considerable increase in feature searched. The rate of feature selection over all the experiments seemed to stay constant at approximately 21%.

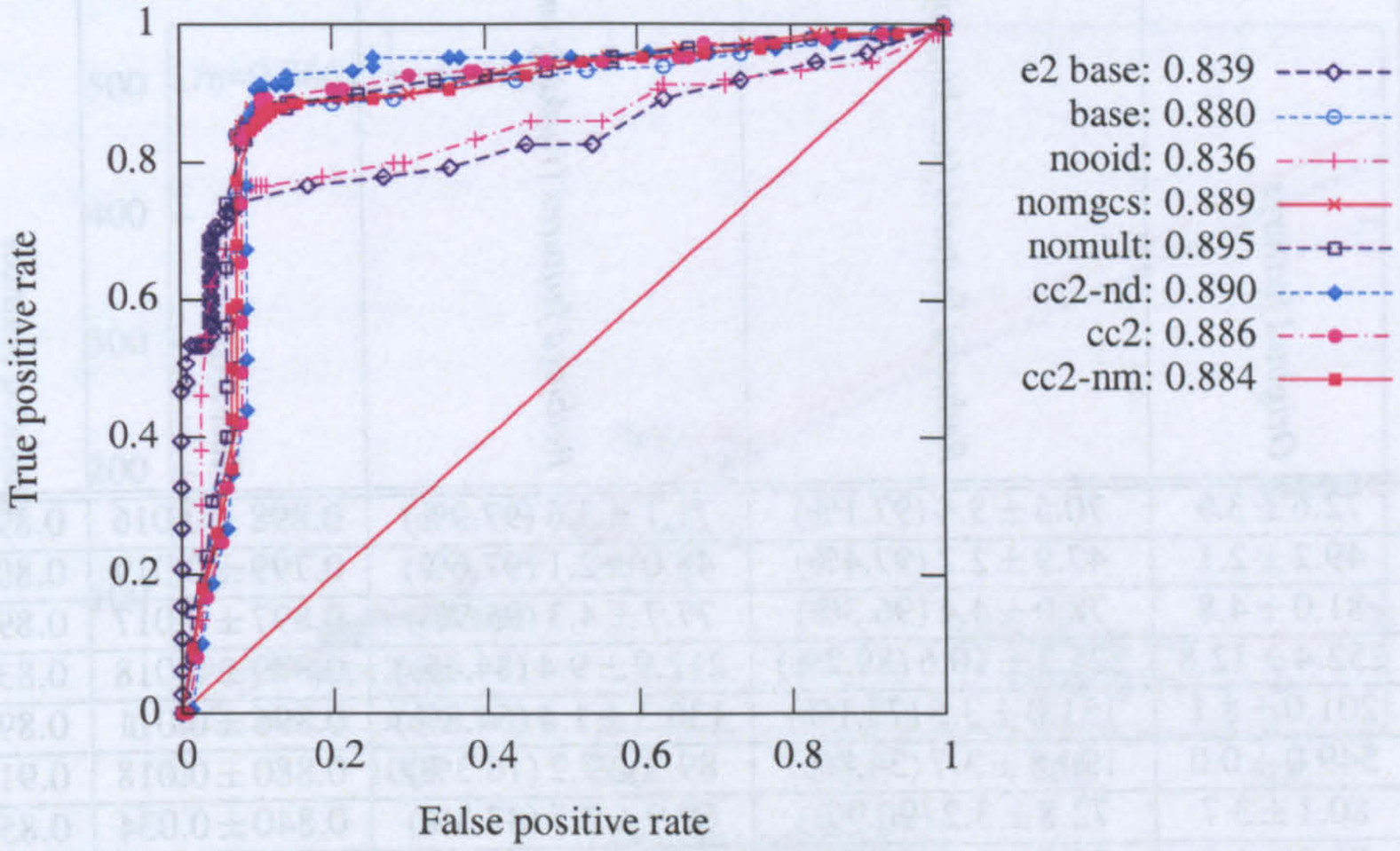


Figure 7.9: Combined ROC curve for CoSINUS variants

We revisit the graphs used in the previous section to visualise some of these results. Figure 7.8(a) shows a similar favourable result in terms of the number of features searched and the accuracy, as again, despite the removal of important aspects of the object model in some of the learners, the points representing CoSINUS experiments again all appear on the convex hull of the experiments depicted. The point representing the removal of object identity again shows a considerable drop in predictive performance. The results for clause combinations of length 2 dominate the highly-performing experiments, with the variant disabling abstract and disjoint classes appearing to be optimal. The points for each experiment in figure 7.8(a) demonstrate a similar rate of feature selection for each variant learner, since they are approximately colinear with each other and the origin. The points corresponding to combination classes of length 2 seem to suggest that feature selection is more strict for these experiments, or alternatively, that the learners using combination classes of length 2 require a slightly larger search for each selected feature.

Finally, we consider the ROC curves for the variant learners. The PROGOL experiment for the base E_2 learner is repeated here for comparison. Each CoSINUS learner, not including the object identity variant, follows approximately the shape described for CoSINUS curves in figure 7.9. Interestingly, the omission of object identity from the model yields a broadly similar curve to that of the base learner using the E_2 form of the mapping. This may suggest that theories induced without the object identity bias result in a theory with overly-specific rules and for which too many examples are covered by the default rule.

7.5.3 Assessing the effect of REFER on propositionalised data

Having completed the comparative analysis of CoSINUS and PROGOL, we now consider empirical results regarding the REFER algorithm. One of the principal claims of REFER *is that it is able to remove logically* redundant features from a given dataset without significantly changing the accuracy of a classifier using it. We report data in this section which suggests this claim, using the results obtained from propositional learning problem transformed by CoSINUS's propositionalisation. Of interest is the degree to which REFER reduces

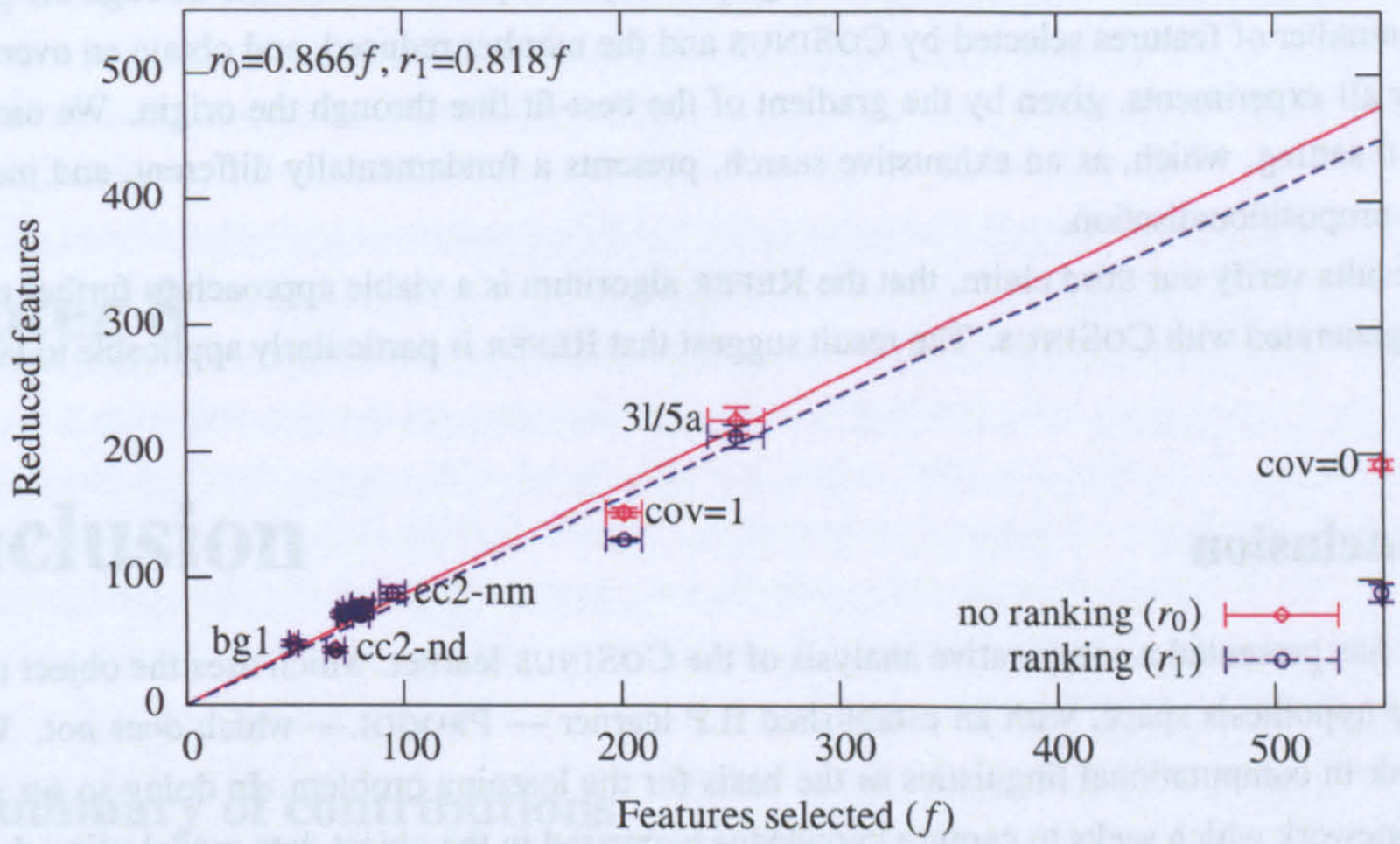
Experiment	*Original features*	*Reduced features (ranking off)*	*Reduced features (ranking on)*	*Test AUC*	*Test AUC without REFER*
base	72.6 ± 3.9	70.5 ± 3.4 (97.1%)	71.1 ± 3.6 (97.9%)	0.898 ± 0.016	0.894 ± 0.014
bg1	49.2 ± 2.1	47.9 ± 2.2 (97.4%)	48.0 ± 2.1 (97.6%)	0.799 ± 0.019	0.803 ± 0.015
2l/5a	81.0 ± 4.8	78.0 ± 4.4 (96.3%)	77.7 ± 4.3 (95.9%)	0.897 ± 0.017	0.895 ± 0.017
3l/5a	252.4 ± 12.8	225.2 ± 10.6 (89.2%)	212.9 ± 9.4 (84.4%)	0.879 ± 0.018	0.830 ± 0.019
cov=1	201.0 ± 8.1	151.0 ± 3.5 (75.1%)	130.3 ± 1.4 (64.8%)	0.896 ± 0.011	0.890 ± 0.014
cov=0	549.0 ± 0.0	190.8 ± 3.7 (34.8%)	89.3 ± 7.2 (16.3%)	0.880 ± 0.018	0.918 ± 0.014
nooid	80.1 ± 3.7	72.8 ± 3.2 (90.9%)	69.9 ± 2.7 (87.3%)	0.840 ± 0.034	0.851 ± 0.035
nomgcs	71.5 ± 1.9	70.5 ± 1.6 (98.6%)	70.6 ± 1.7 (98.7%)	0.900 ± 0.016	0.900 ± 0.016
nomult	80.5 ± 4.9	76.4 ± 3.8 (94.9%)	76.8 ± 4.0 (95.4%)	0.893 ± 0.019	0.896 ± 0.018
cc2	76.3 ± 4.4	74.7 ± 3.9 (97.9%)	74.3 ± 3.8 (97.4%)	0.905 ± 0.014	0.905 ± 0.014
cc2-nd	68.2 ± 4.4	43.1 ± 2.0 (63.2%)	42.7 ± 2.0 (62.6%)	0.915 ± 0.020	0.915 ± 0.020
cc2-nm	94.6 ± 6.1	87.5 ± 5.1 (92.5%)	87.3 ± 5.3 (92.3%)	0.895 ± 0.020	0.899 ± 0.017

Table 7.8: Summary of results for REFER for COSINUS experiments

the feature set, and whether there is a significant change in AUC between the theories induced on the datasets before and after reduction by REFER.

Table 7.8 summarises the results obtained during each experiment. Feature set sizes for the original transformed feature set and reduced feature sets are reported, as well as the test AUC on the theory induced before and after application of REFER. Some of this information is reported graphically. Figure 7.10 shows the reduction in the number of features graphically. Each point on the scatter plot represents a single experiment, showing the relation between the number of original features and the features resulting from applying REFER with and without feature ranking. We demonstrate REFER only in the context of this propositionalisation learner. In particular, PROGOL runs are omitted due to the fact that there is is no intermediate propositionalisation stage in PROGOL. Previous results have suggested that the performance of REFER is naturally due to the amount of logical redundancy present in the data. Results on general datasets as well as propositionalised ones may be found in [5].

Considering first the data, we notice that for many of the datasets, there was relatively little reduction, in many cases yielding a feature set size of around 95% of the original. We discussed before that in these experiments, COSINUS demonstrates a selective approach to feature construction. Although in these cases the reduction offered by REFER was slight, it could also point to the fact that the method adopted produces datasets with relatively little logical redundancy. This may be due in part to the strict bias assumed by the COSINUS learner. In other experiments, feature reduction was considerable. This occurs in two particular settings — that in which the coverage bound is set to be very low, as in experiment *cov=1* and also in the exhaustive search undertaken by experiment *cov=0*. Necessarily, both these settings result in relatively large numbers of



The unlabelled cluster of points correspond to experiments *nomgcs*, *base*, *cc2*, *nooid*, *nomult* and *2l/5a*. The experiment *cov=0* is not included in the best-fit line.

Figure 7.10: Feature reduction by REFER on the propositionalised data.

selected features. The experiment *cov=0* generates all possible features constructable under the metaknowledge supplied. In this set there are necessarily features which are logically redundant. This is particularly relevant to the object model. For instance, consider two simple features which differ only in one class constraint which are evaluated for three examples, p_1 and p_2, positive examples, and n_1, a negative example. Consider that f constrains a term to be of class c while g constrains it to class c'. If the relational part is such that n_1 fails for all substitutions of its variable and p_1 and p_2 succeed, the set of examples in which f is true must necessarily be a subset of those for which g is true, with the negative examples being false. The feature f is then necessarily redundant. REFER effectively selects points on the class hierarchy by which to constrain. This effect is seen strongly in the result for *cov=0*, in which a reduction to 34.8% of the original data is seen. With feature ranking enabled, an even more dramatic gain is seen, to 16.3%. Similar, though not as dramatic, results are reported for *cov=1*. This suggests that the choice of coverage bound of 3 has already performed much of the reduction of logically redundant features during search. In general, REFER performs better on larger datasets, partly because there are more features against which existing ones can be found to be logically redundant. The *3l/5a* (maximum of 3 literals and 5 applications of the refinement operator) shows this, with greater gains for the feature ranking technique. A final point of interest is the result for the learner using no object identity. This result suggests that the adoption of object identity during search reduces the number of logically-redundant features considered.

The results also show that where the REFER algorithm is applicable, the ordering of its features using feature ranking is beneficial, often leading to an appreciable further reduction. Finally, the test AUC before and after the application of REFER are reported. In most cases they are very similar, or the application of REFER yields a small increase in the AUC. This would support the suggestion that the experiments involving five refinements may lead to overfitting, since the removal of redundant features gives the most appreciable increase in this case.

Figure 7.10 summarises the feature reduction graphically. We plot a best-fit line through the points representing the number of features selected by CoSinus and the number reduced, and obtain an overall score for Refer over all experiments, given by the gradient of the best-fit line through the origin. We omit from this fit the *cov=0* setting, which, as an exhaustive search, presents a fundamentally different, and incomparable, approach to propositionalisation.

These results verify our third claim, that the Refer algorithm is a viable approach to further reducing the feature sets generated with CoSinus. The result suggest that Refer is particularly applicable to larger feature sets.

7.6 Conclusion

This chapter has presented a comparative analysis of the CoSinus learner, which uses the object model in its search of the hypothesis space, with an established ILP learner — Progol— which does not. We adopted a relevant task in computational linguistics as the basis for the learning problem. In doing so we presented a mapping framework which seeks to capture knowledge expressed in the object data model using the types and modes bias typical of more conventional inductive logic programming systems. This mapping framework intended to closely represent aspects of the object model while maintaining a natural representation for the Prolog data. The broad application area of natural language processing was presented, in particular the relevance of object-orientation in modelling grammatical structures and a summary of how natural language problems have been solved with inductive logic programming. We reviewed the meaning of a corpus and the specific structure of the Susanne corpus, paying particular attention to how they are respresented relationally. The notion of dependency between grammatical elements — nodes in a parse tree — was then reviewed.

Several key claims of the CoSinus learner were empirically demonstrated, first on various parameter settings for CoSinus and secondly on a series of possible variants of the CoSinus learner which make use of restricted subsets of the object model. Firstly, we argued that CoSinus uses the object metaknowledge to perform a more efficient search of the hypothesis space. The CoSinus learner consistently produced rulesets which were better-performing than its Progol counterpart, even for variants of CoSinus which had aspects of its data model disabled. Considering this predictive performance, CoSinus constructed featuresets which were unusually small, consisting of a few hundred features at most. We showed that for all settings adopted, CoSinus represented the best solution for any tradeoff between AUC and the number of features searched. Additionally, we assessed the degree to which features searched by Progol were metaknowledge-invalid under the mapping adopted, and argued that the use of domain-specific metaknowledge aided the search.

The study also provided an empirical evaluation of the Refer algorithm on datasets propositionalised from the runs of CoSinus. Refer was shown to be particularly effective on larger datasets, while not affecting the AUC of the resulting theory. Additionally, the utility of the ranking process in Refer which first orders features according to a coverage score, was shown to lead to a considerable further reduction over using the feature ordering in the data.

Chapter 8

Conclusion

8.1 Summary of contributions

The core contribution of the thesis is the CoSinus algorithm and its underlying CorLog logical framework. CoSinus is both an algorithm and data mining system for inductive logic programming under the object model, in particular a feature search for propositionalisation using an object-specific refinement operator. CorLog is an extension of an existing object-oriented logical framework which extends the logic programming basis of ILP with a new set of features to guide induction, and define a strong bias, leading to a smaller hypothesis space but still retaining the expressiveness of the object logic. The object model gives rise to a much more sophisticated form of type-safety, involving conjunctions of simple, parametric and abstract classes and constraints on co-existing classes, as well as mechanisms to abstract existing structure in terms of these classes. Relations may take ordered arguments, extending the basis of value restriction in search from classes to constraints involving constants, such as simple integer inequalities. The framework gives rise to redefinition of valid substitution under the new data model, informing a moded, classed refinement operator which has desirable optimality properties, implemented in a three-tiered (feature, rule, theory) search taking advantage of the natural data encapsulation of the object model. Further aspects of the object model are exploited to arrive at search bounds derived from the object model. While there are other algorithms which perform inductive logic programming, and specifically those which propose object-like data mining in class hierarchies, no system known to the author at the time of writing is as comprehensive in its coverage of the object-oriented data model or its use of this data model in constraining induction as closely, and it is in this aspect that much of the innovation lies. Such an approach to induction is immediately useful not only for domains with a strong notion of individual but also where objects naturally belong in a class taxonomy or which adopt complex datatypes modelled independent of their constituents. We suggest that for the de-facto forms of domain description in use in ILP today, such data is inadequately modelled.

The second contribution is the Refer algorithm, which overcomes the often prohibitively high dimensionality of the feature sets produced by propositionalisation. This is of clear interest to the thesis, since the core inductive logic programming technique of CoSinus is propositionalisation. Refer is a post-processing feature reduction stage for feature sets containing Boolean examples labelled by one of any number of possible class labels. Refer is therefore general to the large class of learners producing such data and uses the notion of logical coverage to efficiently determine the logical redundancy of a feature for building a classification rule.

Similar feature reduction algorithms have been proposed. However, the innovation here lies in the partitioning of examples into a set of disjoint subsets whose examples are mutually close in terms of Hamming distance. Comparisons between each of these subsets lead to a greater reduction than comparing the example set as a whole. Its usefulness to the topic of the thesis is that it supplements the redundancy-reducing mechanisms of CoSinus, which use a metaknowledge-based definition of redundancy, and further filters logically redundant hypotheses before presentation to the learner. A number of variants are suggested, and improvements in predictive performance, quality of induced theories and reductions in induction time are demonstrated.

The third main contribution is the application of machine learning to a learning task from the field of computational linguistics. A detailed linguistic corpus is preprocessed with a sophisticated, domain-specific preprocessor to produce a highly-structured representation of thousands of English-language sentences, including their parse trees (the principal compositional structure), grammatical categories (the principal inheritance structure), parts of speech, *etc.*. A complex domain library is defined which allows querying of these sentences in a highly object-oriented manner encompassing the modelling constructs of CorLog and CoSinus as well as augmenting it with additional, derived structure via additional rules. Two inductive logic programming systems, one which uses the object model and one which does not, are used to solve the problem of labelling the nodes within a sentence with its grammatical function. In particular, we study the labelling of a node as the subject of a sentence, or otherwise. Problems of this kind have been approached many times in computational linguistics, but typically by statistical methods, rather than the symbolic, structural approach taken by ILP. Theories expressed in symbolic logic are of interest to linguists, since they directly describe linguistic rules for these important lingustic features. More generally, models for labelling grammatical functions of words are immediately useful for document analysis and retrieval applications. The work tests the claims in the main aim of this thesis in a practical setting by establishing a mapping between the framework of the object model and the conventional logic programming basis and inductive bias of existing ILP learners. Results from applying the learners under various parameter settings and combinations of available background knowledge are compared, showing that object-oriented data mining is a viable approach for real-world problems in computational linguistics, and in particular the labelling of grammatical functions.

8.2 Further work

There are many possible extensions and new directions which this work could take. Some of the most interesting and promising possible directions are discussed here.

8.2.1 Extending the data model

Object-orientation encompasses a wide (and little agreed upon) range of possible data representation elements which have not been explored in this thesis but which could be explored in the future.

For example, we could consider *metaknowledge between methods* Submethods, inverse methods and set-theoretic methods are examples of metaknowledge which exist *between* methods, an aspect of many object models which has not been considered in CoSinus. Inter-method metaknowledge largely allows further detection of redundancy between different but equivalent forms of data expression in a constructed clause. *Bringing other ingredients of ILP into the domain description* extends the notion of including relevant metaknowledge into the domain description to other aspects of the ILP process. Learning problems in ILP may often be put

down to the adoption of an inappropriate cost function, stopping criterion in partial propositionalisation, or other parameters relevant to the needs of the domain. This could be approached by requiring the user to implement parts of the program, included in the domain description, or a library of common cost functions, stopping criteria, and so on.

Generality itself may also be redefined under the object model. *Combining specialisation and generalisation* is a technique used in many ILP learners. Having defined a generality ordering over CORLOG clauses, it would seem profitable to consider search in more than just the general-to-specific direction. Moreover, *other forms of generality* exist under the object model. Object-cube approaches to generalisation consider the class as a set of objects, generalising, for example, an object identifier to the identifier of a class to which it belongs, a class to its superclass, or a set of objects to the multisets of the classes to which they belong, and so on. Alternatively, the inverse resolution approach of several ILP learners may be adapted to the resolution process of F-Logic. Such additional bases for generality orders may prove more successful as an alternative, or in conjunction with, the proposed method. *Expanding on the constraint approach* presented in the ordered methods of COSINUS, situating it within the general field of constraint logic programming while bringing it into closer cohesion with the object model, is likely to further aid learning. The constraint approach presented here is limited in that it considers only expressions of arity 2 and requires declaration of the possible values. These restrictions could feasibly be lifted, and furthermore, further pre-analysis on the values in the database appearing in the constraints could be performed, allowing the learner to determine appropriate points in the constraint hierarchy to test during search. The handling of constraints as presented here necessarily makes the hypothesis space much larger, since it is required that families of constraints are represented as features. An exhaustive search is often impractical, although there are several ways in which this could be overcome, which could provide possible avenues of further investigation. Firstly, the lattice induced by the ordering could be sampled. Secondly, only those conditions associated with a change of class in the training data could be considered. Thirdly, the lattice of possible constraints can be precomputed, and this lattice searched non-sequentially, taking advantage of pruning where appropriate.

8.2.2 Extending the learning task

As well as the object model, we may also consider alternatives and extensions to the learning task. Firstly, we consider *taking the learner beyond classification rules*. Binary classification rules are the traditional setting for ILP. The algorithm is likely to be extendable to other forms of knowledge such as association rules, where classification rules are inappropriate. Among these may be the consideration of *similarity and clustering for objects*. This relies on the adaptation of the object framework to define similarity of objects. This would permit the family of distance-based approach to machine learning, such as clustering. Alternatively, *frequent substructure discovery* is a potentially powerful extension to the algorithm presented. A preprocessing step identifies common structures existing in individuals or parts of individuals, facilitated by the abstraction of structures brought about by using parametric classes. Such frequent substructures may be asserted as new objects to be used in matching individuals during induction. The *incorporation of statistical and probabilistic approaches to object induction* would allow the handling of uncertainty, a shortcoming of many ILP systems. The recent popularity of Bayesian networks for multi-relational data mining has proved very useful in dealing with these shortcomings. With respect to *handling other kinds of data*, such as the handling of noisy or uncertain data or data involving continuous attributes, the algorithm presented inherits shortcomings from ILP. There has been in-

terest in recent years in performing data mining on multimedia data, which itself often contains structure which can be abstracted using the object model, allowing specialised domain libraries for these complex datatypes. Of particular interest is the modelling of examples which are not independent, since assuming example independence, a common assumption of the individual-centred representation, limits the type of model which can be used. Since many domains naturally can not assume example independence, research areas such as link and graph mining which learn from *networks* of examples.

8.2.3 Domain capture and refinement

It has long been an aim of the object-oriented community to make the process of object-oriented analysis more intuitive. The metaknowledge presented exists in the form of relatively low-level logical facts. Capturing domain knowledge is a *human* process, and one best done by a domain expert. Accordingly we identify improvements which would aid and expedite this process of domain capture. In particular, *tools for capturing domain knowledge*, particularly interactive, graphical tools, would facilitate domain modelling by a domain expert, producing metaknowledge declarations for the inductive system. The tool would facilitate the combination of data from differing sources, either by standardised mapping languages or from schema specifications such as OWL or RDFS. Alternatively, *interactive ILP*, in which the domain expert can identify poor features and anomalies, or select between alternative hypotheses, would be an on-line approach to improving learning and refinement of the domain model.

These suggestions concern the definition of the domain description by a domain expert. However, techniques for automatic or semi-automatic analysis of the database for refinement of the domain description may prove beneficial to the inductive task. For example, *increasing the granularity of the class structure* by introducing new classes assists rule induction by introducing a finer-grained hierarchy of classes. The *construction of new methods to aid encapsulation* relates to recent work in which a series of commonly-occuring sequences of method calls can be encapsulated into a new method by similar database analysis, known by the term *macro-operators* [109]. Using these simplified methods is likely to reduce the hypothesis space greatly. Such an approach complements the granularity-increasing techniques. Techniques which *determine the relevancy of methods* aim to overcome the common problem of excessive or irrelevant background knowledge hindering the ILP system. General domain libraries typically offer a wide selection of possible methods and other object modelling features. An on-line analysis module would determine during search which of these are statistically relevant to the task, ensuring a further favourable tradeoff between the restrictiveness and expressiveness of a hypothesis space in which less relevant methods are omitted.

8.2.4 Propositionalisation

A number of directions exist for extending the approach to propositionalisation presented in this thesis.

Extending propositionalisation beyond Boolean features so that the features take values of a given type (class) or denote classes themselves, *i.e.* sets of possible values, has a number of immediate benefits. It takes advantage of the ability of the propositional learner to use non-Boolean data, can be used to express more than just existential features and reduces the dimensionality of the propositionalised dataset by obviating the need for separate boolean features representing, for example, 'the train has one car', 'the train has two cars', and so on. *Better bounds for propositionalisation* are likely to produce a better set of candidate features. At present, a simple coverage bound is adopted. Investigating bounds specific to the object model, or even allowing the

specification of bounds on a per-domain basis, is likely to benefit learning and indirectly reduce overfitting concerns. *Scalability* with respect to dataset size has been a common issue with ILP systems in general. Integration with object databases provides a possible natural and useful means of overcoming scalability issues. Finally, at present the covering approach selects either one rule only (in iterative mode) or all rules (in single mode). The most successful theory may be one which is constructed by taking more than one rule during each iteration. Investigating successful heuristics for rule selection during each propositionalisation step (and possibly the rejection of rules from previous steps) is likely to improve the theory construction process.

8.2.5 Future directions for REFER

The REFER feature reduction algorithm offers many possibilities for future development and analysis. We identify a few of these. *Other means of partitioning the example set in* REFER exist, and while the proposed methods are demonstrably better at reducing feature sets than prior approaches, the partitioning is necessarily simple in order to maintain favourable computational time limits. Chapter 5 presented an analysis of REFER, characterising the properties of partitions upon which it performs well. A simple partitioning scheme maximising these properties would likely benefit feature reduction greatly. *Probablistic* REFER may be an interesting future solution to noise and uncertainty in propositionalised datasets by extending logical feature coverage from the subset relation to a probabilistic definition. Feature searches in probabilistic domains, for example as a result of using stochastic logic programming, may be be performed against a coverage threshold. We can consider combining REFER *and the object model.* In this approach taken in this thesis, the processes of propositionalisation/induction and the feature reduction on the resulting data are highly decoupled; indeed, REFER assumes nothing more than the construction of classification rules from its features. REFER could be feasibly optimised for the object model, by introducing feature costs, pre-grouping features according to some aspect of the data model, or by adopting a distance measure based on the similarity of objects. Correspondences between REFER and formal concept analysis [53] may indicate appropriate strategies for linking the two together. Finally, there may be opportunities for *applying* REFER *during the search process* instead of as a post-processing step. Measures built in to the notions of clause validity and the object model in general reduce redundancy *within the feature* over the traditional approaches to ILP, but REFER offer another criterion for the detection of logical redundancy among sets of features. In the approach presented in this thesis, the reduction is performed as a post-propositionalisation step. By bringing a (possibly simplified variant of) REFER into the search process, further redundancies between sets of features become detectable not necessarily related to syntactic subsumption, although this redundancy is only detected on the basis of the examples in the training set. Accordingly, common sets of features generated during search, such as those under a node of the refinement tree, those corresponding to a particular constituent refinement operator, or even the set of features so far generated, can be analysed using REFER and eliminated — or even pruned — if found redundant. Such redundancy could possibly be incorporated into the substitution framework directly.

8.2.6 Additional application domains

Finally, while useful conclusions were drawn from the application domain studied, it would be instructive and interesting to take such work further. *Comparison with other ILP learners and forms of inductive bias* beyond PROGOL would further aid comparison between the COSINUS learner (utilising various combinations of available metaknowledge) and other ILP learners, specifically in terms of their search strategies, utilisation

of available bias, and other tests relating to the aims of the thesis. There is scope to *apply the algorithm to other domains*. There are a wide variety of domains which naturally suit the object model. Particular application areas which ILP have been applied to in recent years include: the world-wide web and particularly the semantic web; bioinformatics and computational biology; social network analysis and counter-terrorism; and geospatial data. Much of this data involves dependencies between examples, however. Many further applications exist within the studied area of computational linguistics and information extraction. With respect to language itself, the detection of interdependency between nodes in the parse tree may be of interest as well as labelling of other grammatical information than functions.

8.3 Final conclusions

We have presented COSINUS, a technique for data mining in databases which adopts the object oriented data model, and CORLOG, its underlying logical language. COSINUS is an ILP learner which uses a refinement operator to select features for propositionalisation. By adopting a complex notion of class and type correctness, abstraction of structure, method metaknowledge and constraint-based search, new forms of subsumption and substitution are defined and classification rules are induced which utilise the object model. Many domains suitable for ILP possess properties such as the strong notion of (structured) individual, natural taxonomies of objects at multiple levels of granularity, complex or constituent classes with their own semantics, default reasoning, *etc.* which may be exploited for induction but are nevertheless ignored by many inductive learners or beyond their capabilities. For a given domain, the proposed algorithm allows the definition of these aspects of the object model, improving the size/expressiveness tradeoff in the hypothesis space, removing redundant features from the search, and giving better predictive performance than prior ILP learners under comparable parameter settings.

Propositionalisation produces large feature sets, negatively impacting the predictive performance and running times of learners. A means of overcoming these issues in the general-purpose algorithm REFER by efficiently eliminating features from Boolean datasets which are logically redundant for classification by constructing partitions of the example set was presented, analysed and demonstrated.

We successfully demonstrated these properties of the system on a real-world document analysis task using a detailed, heavily preprocessed corpus. By using the object-oriented data model for representation and induction, a better set of hypotheses is searched and more comprehensible models with better predictive accuracy are produced compared to an existing, established ILP learner.

Bibliography

[1] Hassan Aït-Kaci, Bruno Dumant, Richard Meyer, Andreas Podelski, and Peter Van Roy. *The Wild LIFE Handbook*. Digital Equipment Corporation, March 1994.

[2] Hassan Aït-Kaci and R. Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[3] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. In *Proceedings on 3rd International Symposium on Programming Language Implementation and Logic Programming*, pages 255–274, Berlin, 1991.

[4] Hussan Aït-Kaci. *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially Ordered Types*. PhD thesis, University of Pennsylavania, 1984.

[5] Annalisa Appice, Michelangelo Ceci, Simon Rawles, and Peter A. Flach. Redundant feature elimination for multi-class problems. In Russ Greiner and Dale Schuurmans, editors, *Proceedings of the 21st International Conference on Machine Learning (ICML'2004)*, pages 33–40, Banff, Alberta, Canada, July 2004. ACM.

[6] Guillermo Arango. Domain analysis: From art form to engineering discipline. *SIGSOFT Engineering Notes*, 14(3):152–159, May 1989.

[7] Liviu Badea and Shan-Hwei Nienhuys-Cheng. A refinement operator for description logics. In James Cussens and Alan M. Frisch, editors, *Inductive Logic Programming: Proceedings of the 10th International Conference*, volume 1866 of *Lecture Notes in Computer Science*, pages 40–59. Springer, 2000.

[8] Liviu Badea and Monica Stanciu. Refinement operators can be (weakly) perfect. In Sašo Džeroski and Peter A. Flach, editors, *Inductive Logic Programming, 9th International Workshop, ILP-99, Bled, Slovenia, June 24-27, 1999, Proceedings*, volume 1634 of *Lecture Notes in Computer Science*, pages 21–32. Springer, 1999.

[9] Mira Balaban. The F-Logic Approach for Description Languages. Technical report, Ben-Gurion University of the Negev, 1993.

[10] Hendrik Blockeel, Luc de Raedt, and Jan Ramon. Top-down induction of clustering trees. In Jude W. Shavlik, editor, *Proceedings of the 15th International Conference on Machine Learning*, pages 55–63. Morgan Kaufmann, 1998.

[11] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, 2nd edition, 1994.

[12] Henrik Boström. Induction of recursive transfer rules. In James Cussens and Sašo Džeroski, editors, *Learning Language in Logic*, volume 1925 of *Lecture Notes in Computer Science*, pages 237–246. Springer, 1999.

[13] Wray L. Buntine. Generalized subsumption. In J. B. H. du Boulay, David Hogg, and Luc Steels, editors, *Proceedings of the Seventh European Conference on Artificial Intelligence (ECAI '86)*, pages 15–23, July 1986.

[14] Wray L. Buntine. Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36(2):149–176, 1988.

[15] Peter P. Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[16] Weidong Chen and David Scott Warren. C-logic of complex objects. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 369–378. ACM Press, 1989.

[17] Noam Chomsky. Three models for the description of language. *IRI Transactions on Information Theory*, 2(3):113–124, 1956.

[18] Peter Clark and Robin Boswell. Rule induction with CN2: Some recent improvements. In Yves Kodratoff, editor, *Proceedings of the Fifth European Working Session on Learning (EWSL-91)*, pages 151–163. Springer, 1991.

[19] Peter Clark and Tim Niblett. The CN2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[20] Vincent Claveau, Pascale Sébillot, Cécile Fabre, and Pierrette Bouillon. Learning semantic lexicons from a part-of-speech and semantically tagged corpus using inductive logic programming, August 2003.

[21] William W. Cohen. Rapid prototyping of ILP systems using explicit bias. In Francesco Bergadano, Luc de Raedt, Stan Matwin, and Stephen Muggleton, editors, *Proceedings of the IJCAI-93 Workshop on Inductive Logic Programming*, pages 24–35. Morgan Kaufmann, 1993.

[22] Diane J. Cook and Lawrence B. Holder. Substructure discovery using minimum description length and background knowledg. *Journal of Artificial Intelligence Research (JAIR)*, 1:231–255, 1994.

[23] James Cussens. Part-of-speech disambiguation using ILP. Technical Report PRG-TR-25-96, Oxford University Computing Laboratory, UK, 1996.

[24] James Cussens, Sašo Džeroski, and Tomaz Erjavec. Morphosyntactic tagging of Slovene using Progol. In Sašo Džeroski and Peter A. Flach, editors, *Proceedings of the Ninth International Workshop on Inductive Logic Programming (ILP'99)*, volume 1634 of *Lecture Notes in Computer Science*, pages 68–79, Bled, Slovenia, July 1999. Springer.

[25] James Cussens and Stephen G. Pulman. Experiments in inductive chart parsing. In James Cussens and Sašo Džeroski, editors, *Learning Language in Logic*, volume 1925 of *Lecture Notes in Computer Science*, pages 143–156. Springer, 2000.

[26] Claudia d'Amato, Nicola Fanizzi, and Floriana Esposito. A semantic similarity measure for expressive description logics. In *Convegno Italiano di Logica Computazionale (CILC 2005)*, June 2005.

[27] Luc de Raedt. *Interactive Theory Revision: An Inductive Logic Programming Approach*. Academic Press, London, UK, 1992.

[28] Luc de Raedt. Attribute-value learning versus inductive logic programming: The missing links (extended abstract). In David Page, editor, *Proceedings of the Eighth International Workshop on Inductive Logic Programming*, volume 1446 of *Lecture Notes in Computer Science*, pages 1–8, Madison, Wisconsin, USA, July 1998. Springer.

[29] Luc de Raedt. *Logical and Relational Learning: From ILP to MRDM*. Springer, 2006.

[30] Luc de Raedt and Maurice Bruynooghe. A theory of clausal discovery. In Ruzena Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1058–1063. Morgan Kaufmann, August and September 1993.

[31] Luc de Raedt and Luc Dehaspe. Clausal discovery. *Machine Learning*, 26(2–3):99–146, 1997.

[32] Luc de Raedt and Maurice Druynooghe. An overview of the interactive concept-learning and theory revisor CLINT. In Steven Muggleton, editor, *Inductive Logic Programming*, pages 163–191. Academic Press, 1992.

[33] Luc de Raedt and Sašo Džeroski. First-order *jk*-clausal theories are PAC-learnable. *Journal of Artificial Intelligence*, 70(1–2):375–392, 1994.

[34] Stefan Decker, Michael Erdmann, Dieter Fensel, and Rudi Studer. Ontobroker: Ontology based access to distributed and semi-structured information. In Robert Meersman, Zahir Tari, and Scott M. Stevens, editors, *Database Semantics - Semantic Issues in Multimedia Systems, IFIP TC2/WG2.6 Eighth Working Conference on Database Semantics (DS-8)*, volume 138 of *IFIP Conference Proceedings*, pages 351–369. Kluwer, January 1999.

[35] Luc Dehaspe and Luc de Raedt. DLAB: A declarative language bias formalism. In *International Symposium on Methodologies for Intelligent Systems*, pages 613–622, 1996.

[36] Luc Dehaspe and Hannu Toivonen. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36, 1999.

[37] Sašo Džeroski and Tomaž Erjavec. Induction of Slovene nominal paradigms. In Nada Lavrač and Sašo Džeroski, editors, *Proceedings of the Seventh International Workshop on Inductive Logic Programming (ILP'97)*, volume 1297 of *Lecture Notes in Computer Science*, pages 141–148, Prague, Czech Republic, September 1997. Springer.

[38] Sašo Džeroski and Nada Lavrač. An introduction to inductive logic programming. In Sašo Džeroski and Nada Lavrač, editors, *Relational Data Mining*, chapter 3, pages 48–73. Springer, 2001.

[39] Érick Alphonse and Céline Rouveirol. Lazy propositionalisation for relational learning. In *Proceedings of the Fourteenth European Conference on Artificial Intelligence*, pages 256–260. IOS Press, Amsterdam, 2000.

[40] Floriana Esposito, Angela Laterza, Donato Malerba, and Giovanni Semeraro. Refinement of Datalog programs. In *Proceedings of the MLnet Familiarization Workshop on Data Mining with Inductive Logic Programing*, pages 73–94, 1996.

[41] Nicola Fanizzi, Luigi Iannone, Ignazio Palmisano, and Giovanni Semeraro. Concept formation in expressive description logics. In Jean-Francois Boulicaut, Floriana Esposito, Fosca Giannotti, and Dino Pedreschi, editors, *The 15th European Conference on Machine Learning (ECML) and the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, volume 3201 of *Lecture Notes in Computer Science*, pages 99–110. Springer, September 2004.

[42] Usama Fayyad. Knowledge discovery in databases: An overview. In *Relational Data Mining*, pages 28–45. Springer-Verlag New York, New York, NY, USA, 2000.

[43] Peter A. Flach. *Simply Logical: Intelligent Reasoning by Example*. John Wiley & Sons Ltd., Chichester, England, 1997.

[44] Peter A. Flach. From extensional to intensional knowledge: Inductive logic programming techniques and their application to deductive databases. In B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors, *Transactions and Change in Logic Databases*, volume 1472 of *Lecture Notes in Computer Science*, pages 356–387. Springer-Verlag, December 1998.

[45] Peter A. Flach. Knowledge representation for inductive learning. In A. Hunter and S. Parsons, editors, *Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty (ECSQARU)*. Springer, 1999.

[46] Peter A. Flach and Nicholas Lachiche. 1BC: A first-order Bayesian classifier. In Sašo Džeroski and Peter A. Flach, editors, *Ninth International Workshop on Inductive Logic Programming (ILP'99)*, volume 1634 of *Lecture Notes in Artificial Intelligence*, pages 92–103. Springer, June 1999.

[47] Peter A. Flach and Nicolas Lachiche. Confirmation-guided discovery of first-order rules with Tertius. *Machine Learning*, 42(1/2):61–95, January 2001.

[48] Peter A. Flach and Nada Lavrač. Learning in clausal logic: A perspective on inductive logic programming. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond (Essays in Honour of Robert A. Kowalski)*, volume 2407 of *Lecture Notes in Artificial Intelligence*, pages 437–471. Springer, June 2002.

[49] Martin Fowler and Kendall Scott. *UML Distilled*. Addison-Wesley, 1997.

[50] William J. Frawley, Gregory Piatetsky-Shapiro, and Christopher J. Matheus. Knowledge discovery in databases: An overview. In Gregory Piatetsky-Shapiro and William Frawley, editors, *Knowledge Discovery in Databases*, pages 1–30. AAAI/MIT Press, 1991.

[51] Alan M. Frisch. Sorted downward refinement: Building background knowledge into a refinement operator for inductive logic programming. In Sašo Džeroski and Peter A. Flach, editors, *9th International Conference on Inductive Logic Programming (ILP '99)*, volume 1634, pages 104–115, June 1999.

[52] Johannes Fürnkranz and Peter A. Flach. ROC 'n' rule learning — towards a better understanding of covering algorithms. *Machine Learning*, 58(1):39–77, January 2005.

[53] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, Berlin, 1998.

[54] Nicolas Helft. Inductive generalization: A logical framework. In *Progress in Machine Learning: Proceedings of the Second European Working Session on Learning (EWSL '87)*, pages 149–157, Wilmslow, UK, 1987. Sigma Press.

[55] Nicolas Helft. Induction as nonmonotonic inference. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR '89)*, pages 149–156, 1989.

[56] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for expressive description logics. In Harald Ganzinger, David A. McAllester, and Andrei Voronkov, editors, *Logic Programming and Automated Reasoning*, volume 1705 of *Lecture Notes in Computer Science*, pages 161–180. Springer, 1999.

[57] Tamás Horváth, *Zoltán Alexin*, Tibor Gyimóthy, and Stefan Wrobel. Application of different learning methods to Hungarian part-of-speech tagging. In Sašo Džeroski and Peter A. Flach, editors, *Proceedings of the Ninth International Workshop on Inductive Logic Programming (ILP'99)*, volume 1634 of *Lecture Notes in Computer Science*, pages 128–139, Bled, Slovenia, July 1999. Springer.

[58] Daniel Jurafsky and James H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition*. Prentice Hall series in artificial intelligence. Prentice Hall, 2000.

[59] Aram Karalič and Ivan Bratko. First-order regression. *Machine Learning*, 26(2–3):147–176, 1997.

[60] Alan C. Kay. The early history of Smalltalk. *ACM SIGPLAN Notices*, 28(3):69–95, March 1993.

[61] Dimitar Kazakov. Combining LAPIS and WordNet for learning of LR parsers with optimal semantic constraints. In Sašo Džeroski and Peter A. Flach, editors, *Proceedings of the Ninth International Workshop on Inductive Logic Programming (ILP'99)*, volume 1634 of *Lecture Notes in Computer Science*, pages 140–151, Bled, Slovenia, July 1999. Springer.

[62] Dimitar Kazakov, James Cussens, and Surest Manandhar. On the duality of semantics and syntax: The PP attachment case. Technical Report YCS-409, Department of Computer Science, University of York, UK, 2006.

[63] Dimitar Kazakov and Suresh Manandhar. Unsupervised learning of word segmentation rules with genetic algorithms and inductive logic programming. *Machine Learning*, 43(1/2):121–162, 2001.

[64] Dimitar Kazakov, Suresh Manandhar, and Tomaz Erjavec. Learning word segmentation rules for tag prediction. In Sašo Džeroski and Peter A. Flach, editors, *Proceedings of the Ninth International Workshop on Inductive Logic Programming (ILP'99)*, volume 1634 of *Lecture Notes in Computer Science*, pages 152–161, Bled, Slovenia, July 1999. Springer.

[65] Michael Kifer and Georg Lausen. F-logic: A higher-order language for reasoning about objects, inheritance, and scheme. In James Clifford, Bruce G. Lindsay, and David Maier, editors, *SIGMOD Conference*, pages 134–146. ACM Press, 1989.

[66] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.

[67] Michael Kifer and James Wu. A logic for object-oriented logic programming (Maier's O-logic revisited). In *Proceedings of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 379–393, 1989.

[68] Mathias Kirsten and Stefan Wrobel. Relational distance-based clustering. In David Page, editor, *Proceedings of the Eighth International Conference on Inductive Logic Programming*, volume 1446 of *Lecture Notes In Artifical Intelligence*, pages 261–270. Springer, 1998.

[69] Gilles Klopman. Artifical intelligence approach to structure-activity studies: computer automated structure evalusation of biological activity of organic molecules. *Journal of the American Chemical Society*, 106:7315–7321, 1984.

[70] Gilles Klopman. MultiCASE: A hierarchical computer automated structure evaluation program. *Quantitative Structure Activity Relationships*, 11:176–184, 1992.

[71] Stefan Kramer and Eibe Frank. Bottom-up propositionalization. In *Proceedings of the ILP 2000 Work-in-Progress Tracks*, pages 156–162. Imperial College, London, 2000.

[72] Stefan Kramer, Nada Lavrač, and Peter A. Flach. Propositionalization approaches to relational data mining. In Saso Džeroski and Nada Lavrač, editors, *Relational Data Mining*, pages 262–291. Springer, September 2001.

[73] Stefan Kramer, Bernhard Pfahringer, and Christoph Helma. Stochastic propositionalization of non-determinate background knowledge. In David Page, editor, *Proceedings of the Eighth International Conference on Inductive Logic Programming (ILP)*. Springer-Verlag, 1998.

[74] Mark-A. Krogel, Simon Rawles, Filip Železný, Peter A. Flach, Nada Lavrač, and Stefan Wrobel. Comparative evaluation of approaches to propositionalization. In Tamás Horváth and Akihiro Yamamoto, editors, *Proceedings of the 13th International Conference on Inductive Logic Programming (ILP'2003)*, number 2835 in Lecture Notes in Computer Science, pages 197–214, Szeged, Hungary, October 2003. Springer Verlag.

[75] Mark-A. Krogel and Stefan Wrobel. Transformation-based learning using multirelational aggregation. In Céline Rouveirol and Michèle Sebag, editors, *Proceedings of the Eleventh International Conference on Inductive Logic Programming (ILP)*. Springer, 2001.

[76] Mark-André Krogel. *On Propositionalization for Knowledge Discovery in Relational Databases*. PhD thesis, Otto-von-Guericke-Universität, Magdeburg, Germany, July 2005.

[77] Nada Lavrač and Sašo Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.

[78] Nada Lavrač and Sašo Džeroski. LINUS: Using attribute-value learners in an ILP framework. In *Inductive Logic Programming: Techniques and Applications*, chapter 5, pages 81–122. Ellis Horwood, 1994.

[79] Nada Lavrač, Sašo Džeroski, and Ivan Bratko. Handling imperfect data in inductive logic programming. In *Advances in Inductive Logic Programming*, pages 48–64. IOS, 1996.

[80] Nada Lavrač, Sašo Džeroski, and Marko Grobelnik. Learning nonrecursive definitions of relations with LINUS. In Yves Kodratoff, editor, *Proceedings of the Fifth European Working Session on Learning*, pages 265–281. Springer, 1991.

[81] Nada Lavrač and Peter A. Flach. An extended transformation approach to inductive logic programming. *ACM Transactions on Computational Logic*, 2(4):458–494, 2001.

[82] Nada Lavrač, Peter A. Flach, and Blaž Zupan. Rule evaluation measures: A unifying view. In Sašo Džeroski and Peter A. Flach, editors, *The Ninth International Workshop on Inductive Logic Programming (ILP'99)*, volume 1634 of *Lecture Notes in Computer Science*, pages 174–185, Bled, Slovenia, 1999. Springer.

[83] Nada Lavrač, Dragan Gamberger, and Viktor Jovanoski. A study of relevance for learning in deductive databases. *Journal of Logic Programming*, 40(2–3):215–249, 1999.

[84] Nada Lavrač, Filip Železný, and Peter A. Flach. RSD: Relational subgroup discovery through first-order feature construction. In Stan Matwin and Claude Sammut, editors, *Proceedings of the Twelfth International Conference on Inductive Logic Programming (ILP)*, volume 2583 of *Lecture Notes in Computer Science*. Springer, 2002.

[85] Dekang Lin. PRINCIPAR — an efficient, broad-coverage, principle-based parser. In *Proceedings of the fifteenth International Conference on Computational Linguistics (COLING'94)*, pages 42–488, Kyoto, Japan, 1994.

[86] Nikolaj Lindberg and Martin Eineborg. Improving part of speech disambiguation rules by adding linguistic knowledge. In Saso Dzeroski and Peter A. Flach, editors, *Proceedings of the Ninth International Workshop on Inductive Logic Programming (ILP'99)*, volume 1634 of *Lecture Notes in Computer Science*, pages 186–197, Bled, Slovenia, July 1999. Springer.

[87] Francesca A. Lisi. *An ILP Setting for Object-Relational Data Mining*. PhD thesis, Department of Computer Science, University of Bari, December 2002.

[88] Francesca A. Lisi and Floriana Esposito. Ilp meets knowledge engineering: A case study. In *ILP*, pages 209–226, 2005.

[89] Bertram Ludäscher, Rainer Himmeröder, Georg Lausen, Wolfgang May, and Christian Schlepphorst. Managing semistructured data with FLORID: A deductive object-oriented perspective. *Information Systems*, 23(8):589–613, 1998.

[90] D. Maier. A logic for objects. In *Proceedings of the Workshop on Foundations of Deductive Databases and Logic Programming (preprint)*, pages 6–26, Washington, DC, 1986.

[91] Heikki Mannila. Aspects of data mining. In Yves Kodrato, Gholamreza Nakhaeizadeh, and Charles Taylor, editors, *Proceedings of the MLnet Familiarization Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases*, pages 1–6, Heraklion, Crete, 1995.

[92] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: the penn treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[93] Francis McCabe. *Logic and Objects*. Computer Science. Prentice Hall, 1992.

[94] Ryszard S. Michalski. a theory and methodology of inductive learning. *Journal of Artificial Intelligence*, 20(2):111–161, 1983.

[95] George A. Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine Miller. Five papers on WordNet. *International Journal of Lexicology*, 3(4), 1990.

[96] Tom M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, 1982.

[97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[98] Chris Moss. *Prolog++: The Power of Object-Oriented and Logic Programming*. Addison-Wesley, 1994.

[99] Paulo Moura. *Logtalk: Design of an Object-Oriented Logic Programming Language*. PhD thesis, University of Beira Interior, 2003.

[100] Paulo Moura and Ernesto Costa. Logtalk: Object-oriented programming in Prolog (English version). In Amandio Vaz Velho, editor, *Second Portuguese Conferenceand Exhibition on Object-Oriented Technology*, Lisbon, 1994.

[101] Stephen Muggleton. Learning from positive data. In Stephen Muggleton, editor, *Proceedings of the 6th International Workshop on Inductive Logic Programming*, pages 225–244, 1996.

[102] Stephen Muggleton and Michael Bain. Analogical prediction. In Saso Dzeroski and Peter A. Flach, editors, *Proceedings of the Ninth International Workshop on Inductive Logic Programming (ILP'99)*, volume 1634 of *Lecture Notes in Computer Science*, pages 234–244, Bled, Slovenia, July 1999. Springer.

[103] Stephen Muggleton and Wray Buntine. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–352. Morgan Kaufmann, 1988.

[104] Stephen Muggleton and Luc de Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19–20:629–679, 1994.

[105] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, pages 368–391, Tokyo, Japan, 1990.

[106] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In Stephen Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.

[107] Stephen Muggleton and John Firth. Relational rule induction with cprogol4.4: A tutorial introduction. In Sašo Džeroski and Nada Lavrač, editors, *Relational Data Mining*, chapter 7, pages 160–188. Springer, 2001.

[108] Stephen H. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[109] Lourdes Peña Castillo and Stefan Wrobel. Macro-operators in multirelational learning: A search-space reduction technique. In Tapio Elomaa, Heikki Mannila, and Hannu Toivonen, editors, *Thirteenth European Conference on Machine Learning (ECML'02)*, volume 2430 of *Lecture Notes in Computer Science*, pages 357–368. Springer, 2002.

[110] Clare Nédellec, Céline Rouveirol, Hilde Adé, Francesco Bergadano, and Birgit Tausend. Declarative bias in ilp. In Luc de Raedt, editor, *Advances in Inductive Logic Programming, volume 32*, Frontiers in Artificial Intelligence and Applications, pages 83–103. IOS Press, Amsterdam, 1996.

[111] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Artificial Intelligence*. Springer, February 1997.

[112] Natalya F. Noy and Deborah L. McGuinness. Ontology Development 101: A Guide to Creating Your First Ontology. Technical report, Stanford Medical Informatics, Palo Alto, 2001.

[113] Gordon Plotkin. A note on inductive generalization. *Machine Intelligence*, (5):153–163, 1970.

[114] Gordon Plotkin. *Automatic Methods of Inductive Inference*. PhD thesis, Edinburgh University, 1971.

[115] Gordon Plotkin. A further note on inductive generalization. *Machine Intelligence*, (6):101–124, 1971.

[116] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[117] J. Ross Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In Pavel Brazdil, editor, *Proceedings of the Sixth European Conference on Machine Learning (ECML '93)*, number 667 in Lecture Notes in Artificial Intelligence, pages 3–20. Springer-Verlag, 1993.

[118] J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5:135–151, 1970.

[119] Ronald L. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.

[120] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[121] Gian-Carlo Rota. The number of partitions of a set. *American Mathematical Monthly*, 71:498–504, 1964.

[122] Céline Rouveirol. Flattening and saturation: Two representation changes for generalization. *Machine Learning*, 14(1):219–232, 1994.

[123] K. Sagonas, T. Swift, and D.S. Warren. XSB as an efficient deductive database engine. *ACM SIGMOD Conference on Management of Data*, pages 442–453, 1994.

[124] Geoffrey Sampson. *English for the computer: the* SUSANNE *corpus and analytic scheme*. Oxford, 1995.

[125] Y. Sasaki and M. Haruno. RHB$^+$: A type-oriented ILP system learning from positive data. In *IJCAI-97*, pages 894–899, 1997.

[126] Manfred Schmidt-Schauss and Gert Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[127] Giovanni Semeraro, Floriana Esposito, Donato Malerba, Nicola Fanizzi, and Stefano Ferilli. A logic framework for the incremental inductive synthesis of datalog theories. In Norbert E. Fuchs, editor, *Proceedings of the Seventh International Workshop on Logic Program Synthesis and Tranformation*, pages 300–321. Springer, 1998.

[128] Ehud Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusettes, 1982.

[129] Colin Shearer. The CRISP-DM model: The new blueprint for data mining. *Journal of Data Warehousing*, 2000.

[130] Michael Sintek and Stefan Decker. TRIPLE - an RDF query, inference, and transformation language. In *Proceedings of the 14th International Conference on Applications of Prolog*, pages 47–56, University of Tokyo, Tokyo, Japan, October 2001. The Prolog Association of Japan.

[131] Ashwin Srinivasan. Four suggestions and a rule corncerning the application of ILP. In Sašo Džeroski and Nada Lavrač, editors, *Relational Data Mining*, chapter 15, pages 365–374. Springer, 2001.

[132] Ashwin Srinivasan. The ALEPH manual. `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/`, 2007. [Accessed January 2007].

[133] Ashwin Srinivasan and Ross D. King. Feature construction with inductive logic programming: A study of quantitative predictions of biological activity aided by structural attributes. *Data Mining and Knowledge Discovery*, 3(1):37–57, 1999.

[134] Gerd Stumme, Andreas Hotho, and Bettina Berendt. Semantic web mining - state of the art and future directions. *Journal of Web Semantics*, 2006.

[135] Birgit Tausend. Biases and their effects in inductive logic programming. In F. Bergadano and Luc de Raedt, editors, *Proceedings of the Seventh European Conference on Machine Learning*. Springer Verlag, 1994.

[136] Toby J. Teorey, Dongqing Yang, and James P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys (CSUR)*, 18(2):197–222, 1986.

[137] Ljupčo Todorovski, Peter A. Flach, and Nada Lavrač. A report on experiments with weighted relative accuracy in cn2. Technical Report CSTR-00-003, Department of Computer Science, University of Bristol, March 2000.

[138] Paul E. Utgoff and Tom M. Mitchell. Acquisition of appropriate bias for inductive concept learning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 414–417, Los Altos, California, 1982. Morgan Kaufmann.

[139] Ian H. Witten and Eibe Frank. *Data Mining: practical machine learning tools and techniques with Java.* Morgan Kaufmann, 1999.

[140] S. Wrobel. An algorithm for multi-relational discovery of subgroups. In *Proceedings of the First European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD)*, pages 78–87. Springer, 1997.

[141] Guizhen Yang and Michael Kifer. FLORA: Implementing an efficient DOOD system using a tabling logic engine. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *The First International Conference in Computational Logic (CL 2000)*, volume 1861 of *Lecture Notes in Computer Science*. Springer, July 2000.

[142] Filip Zelezny. Efficiency-concious propositionalization for relational learning. *Kybernetica*, 40(3):275–292, 2004.

[143] John M. Zelle, Raymond J. Mooney, and Joshua B. Konvisser. Combining top-down and bottom-up techniques in inductive logic programming. In William W. Cohen and Haym Hirsh, editors, *Proceedings of the 11th International Conference on Machine Learning*, pages 343–351. Morgan Kaufmann, 1994.